ABSTRACT

Title of dissertation: CHARACTERIZING AND ACCELERATING BIOINFORMATICS WORKLOADS ON MODERN MICROARCHITECTURES

Kursad Albayraktaroglu, Doctor of Philosophy, 2007

Dissertation directed by: Professor Manoj Franklin
Department of Electrical and Computer Engineering

Bioinformatics, the use of computer techniques to analyze biological data, has been a particularly active research field in the last two decades. Advances in this field have contributed to the collection of enormous amounts of data, and the sheer amount of available data has started to overtake the processing capability possible with current computer systems. Clearly, computer architects need to have a better understanding of how bioinformatics applications work and what kind of architectural techniques could be used to accelerate these important scientific workloads on future processors.

In this dissertation, we develop a bioinformatic benchmark suite and provide a detailed characterization of these applications in common use today from a computer architect's point of view. We analyze a wide range of detailed execution characteristics including instruction mix, IPC measurements, L1 and L2 cache misses on a real architecture; and proceed to analyze the workloads' memory access characteristics.

We then concentrate on accelerating a particularly computationally intensive bioinformatics workload on the novel Cell Broadband Engine multiprocessor architecture. The

HMMER workload is used for protein profile searching using hidden Markov models, and most of its execution time is spent running the Viterbi algorithm. We parallelize and partition the HMMER application to implement it on the Cell Broadband Engine. In order to run the Viterbi algorithm on the 256KB local stores of the Cell BE synergistic processing units (SPEs), we present a method to develop a fast SIMD implementation of the Viterbi algorithm that reduces the storage requirements significantly. Our HMMER implementation for the Cell BE architecture, Cell-HMMER, exploits the multiple levels of parallelism inherent in this application, and can run protein profile searches up to 27.98 times faster than a modern dual-core x86 microprocessor.

CHARACTERIZING AND ACCELERATING BIOINFORMATICS WORKLOADS ON MODERN MICROARCHITECTURES

by

Kursad Albayraktaroglu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Committee:

Professor Manoj Franklin, Chair/Advisor
Professor Prakash Narayan
Professor Ankur Srivastava
Professor Chau-Wen Tseng
Professor Donald Yeung

DEDICATION

To my parents

ACKNOWLEDGMENTS

First of all, I would like to thank my family for their unconditional love and support. I am also thankful to my wife, Sevim, for her encouragement, her love and her patience through the long and difficult journey of writing a Ph.D thesis while working.

I would like to thank Mohamed Zahran, Renju Thomas, Brinda Ganesh, Michael Black, Aamer Jaleel, Rania Mameesh, David Wang, Sadagopan Srinivasan and many other friends for their friendship and help.

I will be forever indebted to my academic advisor, Professors Manoj Franklin, for believing in me and supporting me throughout my studies at Maryland. I also would like to thank other members of my committee, Dr. Chau-Wen Tseng, Dr. Prakash Narayan, Dr. Donald Yeung and Dr. Ankur Srivastava, for their insights and patience with a dissertation writing process that took place almost entirely away from the campus.

My six-month internship at the IBM T.J. Watson Research Center was the highlight of my doctoral studies. I am grateful to Dr. Michael Perrone for his supervision of my work at T.J. Watson and my subsequent research; and to Dr. Jizhu Lu for his help and insights. I completed the final stages of my graduate studies while working at Advanced Micro Devices (AMD) in Austin, Texas. This dissertation would certainly not have been possible without the tolerance and support of my supervisors, Andy McBride and Bill Chandler. I owe my gratitude to both of them.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

The success of genome sequencing efforts and developments in bioinformatics resulted in a vast amount of data over the last two decades. This growth in data, coupled with increasingly efficient techniques and tools devised by the bioinformatics community, have resulted in many practical applications that are already saving and improving lives. As bioinformatics emerges as an important class of scientific computing applications, it is becoming more evident that further advances in this field can pave the way for economically and scientifically significant achievements in areas such as protein structure prediction for drug discovery and development of gene-based therapies.

We expect that the performance of bioinformatics applications will therefore become an important factor in defining future high performance computing systems. Despite the fact that bioinformatics is a very active research area with many challenging computational performance needs, bioinformatics workloads are not among those typically used by computer architects and systems designers for evaluating new ideas. Our goals in this dissertation are to provide a thorough analysis of the performance characteristics of selected bioinformatics applications from a computer architect's point of view, and to explore avenues to utilize recently proposed novel microarchitectures for accelerating these workloads.

1.1 Problem Definition

A series of high-profile successes have brought bioinformatics to the forefront of modern science in recent years. The best example of these is the successful sequencing of the human genome, which continues to capture the public imagination and increase awareness and interest in bioinformatics research. A more subtle success story in bioinformatics is the amount of genomic data that is being collected. In August 2005, the total amount of data in public DNA and RNA data repositories exceeded 100,000,000,000 bases for the first time [1]. The amount of base pairs in the NCBI GenBank has quintupled between 2000 and 2005, and the average annual growth of this database has been 39.6% over the same period.

However impressive, this rate of growth might look like a mixed blessing if evaluated alongside the growth of microprocessor performance for the same interval. If the current trends continue, annual growth of microprocessor performance might soon be lagging that of genomic data; with potentially important implications on researchers' ability to analyze this data in a timely manner to facilitate future research.

The reasons of the recent slowdown in microprocessor performance growth are complex. The field of computer architecture seems to have arrived at an important crossroads. Due to a combination of factors, time-honored techniques to utilize the increased transistor budgets for higher performance do not seem to work anymore: the performance returns from higher clock frequencies and deeper pipelines are diminishing due to power density limitations and the inability of memory access speeds to keep up with the processor cores [38]. It has been clear for the last several years that future gains in processor

performance had to be obtained by new means; and the industry as a whole started shifting towards new architectural concepts such as simultaneous multithreading (SMT) and chip multiprocessors (CMPs). Recognizing the impracticality of supporting architectural features for many different kinds of workloads, architects have also been revisiting the concept of application-specific accelerators [10]. While the abundance of coarse-grained task parallelism in many bioinformatics applications is well known, there is a need for more detailed workload characterization of these applications in order to optimize their performance on these new architectures, devise new methods of accelerating them, and to define performance goals and suitable architectures for possible accelerator implementations. Equally needed are investigative studies of how recent novel processor architectures could be utilized to accelerate these applications.

Another motivation for more detailed characterization and acceleration of computationally intensive bioinformatics applications is the migration of such workloads to workstations and desktops. The falling costs of disk storage and personal computers has made it possible to run important bioinformatics workloads on researchers' personal workstations. Our consultations with practicing bioinformaticists reveal that applications which once were exclusively run on servers are now increasingly run on personal systems.

With these in mind, we believe that the existing situation necessitates a thorough analysis analysis of bioinformatics workloads (which have some distinct properties differentiating them from traditional scientific workloads included in benchmark suites like SPEC FP, as we will argue in this dissertation) for optimum performance on future microarchitectures, as well as exploration of new architectures for accelerating bioinformatics workloads. Our work in this dissertation attempts to address these issues.

1.2 Contributions and Significance

The contributions of this dissertation are the following:

- We identify and classify a representative set of bioinformatics application benchmarks, and characterize their execution characteristics using hardware performance counters on an x86 processor. Our methodology allows us to study these characteristics over the complete run of each benchmark; and we compare and contrast the statistics to that of SPEC INT and FP benchmark suites.

- Building on the findings of our workload characterization work, we present an analysis of the L2 cache performance characteristics of our bioinformatics benchmarks on several different memory hierarchy configurations chosen to represent future architectures.

- Finally, we consider the possibility of using recent non-conventional processor architectures as a means to accelerate computational biology workloads. As a case study, we use the Cell Broadband Engine (Cell BE) heterogeneous multiprocessor architecture to accelerate protein profile searching, a CPU-intensive application whose performance characteristics have been studied earlier in the dissertation. We describe and discuss the challenges of programming the Cell BE architecture and data distribution strategies; and provide a detailed analysis of our experimental results. Our parallelization method can be used to port similar dynamic programming workloads to the Cell BE.

In order to provide a background for our analysis work, this dissertation also provides a brief introduction to bioinformatics concepts and detailed descriptions of our simulation/-analysis frameworks and the Cell BE multiprocessor architecture.

We believe that our detailed analysis of the execution characteristics of bioinformatics workloads can contribute directly to the evaluation process of similar workloads on future microarchitectures. We also believe that the methodology and results we present for running bioinformatics applications on the Cell BE architecture can contribute to implementations of current and future bioinformatics applications on similar, non-conventional heterogeneous multiprocessor systems.

1.3 Organization of the Dissertation

In this dissertation, we aim to present an overall picture of current bioinformatics applications and gradually expand our analysis to cover microarchitectural implications of these workloads; and eventually present a case study to accelerate one of these workloads on a non-conventional multiprocessor architecture. This first chapter presents a brief overview of our motivation, and outlines the contributions of this dissertation. In the second chapter, we present an introduction to the general concepts of bioinformatics such as introductory molecular biology, the different kinds of data involved, and important application classes. In Chapter 3, we present a core group of bioinformatics workloads emphasizing genomic applications, and proceed to present a detailed workload characterization using hardware performance counters. We describe our methodology and comment on similarities and differences of the execution characteristics of these benchmarks

to those of the SPEC 2000 integer and floating-point benchmarks. In Chapter 4, we focus on the memory system performance of these benchmarks, concentrating on cache access and TLB characteristics. We present the binary instrumentation based cache simulation methodology we utilized, and discuss the implications of our findings. Our studies take a different path in Chapter 5 where we proceed with a case study involving running a CPU-intensive bioinformatics application on a novel multiprocessor architecture: the Cell Broadband Engine (Cell BE). We describe Cell BE and Cell BE programming models briefly; and we present a detailed account of our porting work, complete with discussions of how the unique features of this multiprocessor can be utilized for high performance bioinformatics. Chapter 6 concludes this dissertation with our concluding remarks.

Chapter 2

Introduction to Bioinformatics

Among all the new promising scientific fields, there is little doubt that bioinformatics stands at the center of interest. The promises of using computer technology to decode the human genome to predict gene and protein functions, treat genetic diseases by identifying disease-related genes and rational design of drugs which directly target specific genes and proteins (as opposed to expensive trial-and-error methods) have attracted both big amounts of investment and some of the brightest minds in science to bioinformatics. Never before the arrival of this new science have the fields of computer design and life sciences been so close and so intertwined. In the work that follows, we aim to provide a closer look at common bioinformatics applications from the viewpoint of the computer architect. Before we proceed, a comprehensive introduction to the field of bioinformatics is in order. Some of the questions one might ask are:

- What is bioinformatics? What has led to its emergence?
- What are the most important problems of interest to the bioinformatics community with regards to computer systems performance?
- Just how much and what kind of data are involved? How is this data collected?
- What is the “end product” of bioinformatics, if one may even define such a thing? How are the results obtained through bioinformatics applications used?

- Considering that new drugs and cures are often mentioned as the most promising rewards of the advances in bioinformatics, where does bioinformatics fit in the drug discovery process?

- What are the important bioinformatics application domains?

Along with a general introduction to the complex terminology and basic concepts of bioinformatics, these are the questions that we intend to answer in this introductory chapter.

2.1 What is Bioinformatics?

Bioinformatics can generally be described as the application of information technologies to the classification, organization, and analysis of biological data. At first sight, the combination of biology, the study of living organisms, and information technologies might indeed seem odd. However, as our understanding of the basic building blocks of life gradually increased, the basic building blocks of life start to look more like just data waiting to be discovered and analyzed. From small DNA sequences to complex protein structures, life itself can be viewed as a giant collection of data and its interaction with the world we live in. It probably would not be too farfetched to call life “the original information technology”.

The necessity to introduce information technology into the study of life was a consequence of the sheer amounts of biological data that has been collected by researchers. While such data collection has been going on since the early days of the discovery of DNA, there was a marked acceleration in the rate of data collection in the last couple of

decades. This acceleration, in turn, seems to be a direct result of several new techniques; the most important of which are methods involving high-throughput determination of DNA sequences using ABI sequences, mRNA expression using microarrays, and proteins using mass spectrometry. In addition, the use of robotics technology in biological data collection made it possible to automate tedious, routine procedures that were done manually prior to the invention of these robotic devices. At the same time, the great advances in technologies that were originally developed for VLSI circuit development and production made it possible to produce miniaturized data collection and analysis devices such as the "gene chips" that are used for high-speed gene sequencing. The "perfect storm" that ensued from advances in these technologies and their use in biology, coupled with high-performance microprocessor architectures, shaped the field of molecular biology by significantly increasing the data collection throughput. As computer technology has been introduced to cope with the resulting mountains of data, modern biology and computer technology have become an inseparable pair, creating the field we called *bioinformatics*.

2.2 Basic Biological Concepts

The basic building block of any living organism is the cell. Any living being is made up of cells, which can be very diverse and highly specialized in complex life forms. Cells are considered living organisms themselves, and they interact with their surroundings and other cells through the use of different cellular mechanisms. These interactions make it possible for a cell to sustain its lifecycle, during which it obtains energy, grows, reproduces, and eventually dies. All of these distinct events in the cellular lifecycle are

based on numerous biochemical reactions. These biochemical reactions are controlled primarily by large organic molecules called *proteins*; and these proteins play important roles in vital biochemical processes by triggering them or acting as catalysts. Proteins, along with the overall structure of any cell, are described by the genetic code in the *DNA* (deoxyribonucleic acid) within the cell.

DNA is a macromolecule consisting of two molecules called *DNA strands*. DNA strands are essentially long chains of basic units called nucleotides, which are formed by the combination of one nucleic acid, one sugar, and one phosphate. The different nucleic acids in nucleotides is what encodes genetic data in the DNA. There are four unique nucleic acids types that can exist in a nucleotide. Since each of them can only couple with a certain type of nucleic acid on the other strand, a single DNA strand can sufficiently describe the genetic information encoded on the entire DNA. Possible nucleotides for a DNA strand are Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). A can only form a base pair with T, and C can only pair with G on the other strand. Since the DNA sequence on a strand can consist of any combination of these four nucleotides, this simple structure can encode the genetic information of an immense number of different organisms. Figure 2.1 illustrates the structure of a DNA macromolecule, and highlights the structures of nucleotides and base pairs.

As we mentioned earlier, the vital biochemical functions of cells are only possible through the production of proteins. However, DNA by itself has little role in the actual biochemical workings of the cell-it is the proteins that do all the work. DNA basically contains the ”master plan” of the proteins that are so crucial to the survival of the cell, and also acts as the template used by the cell for reproducing and making copies of itself.

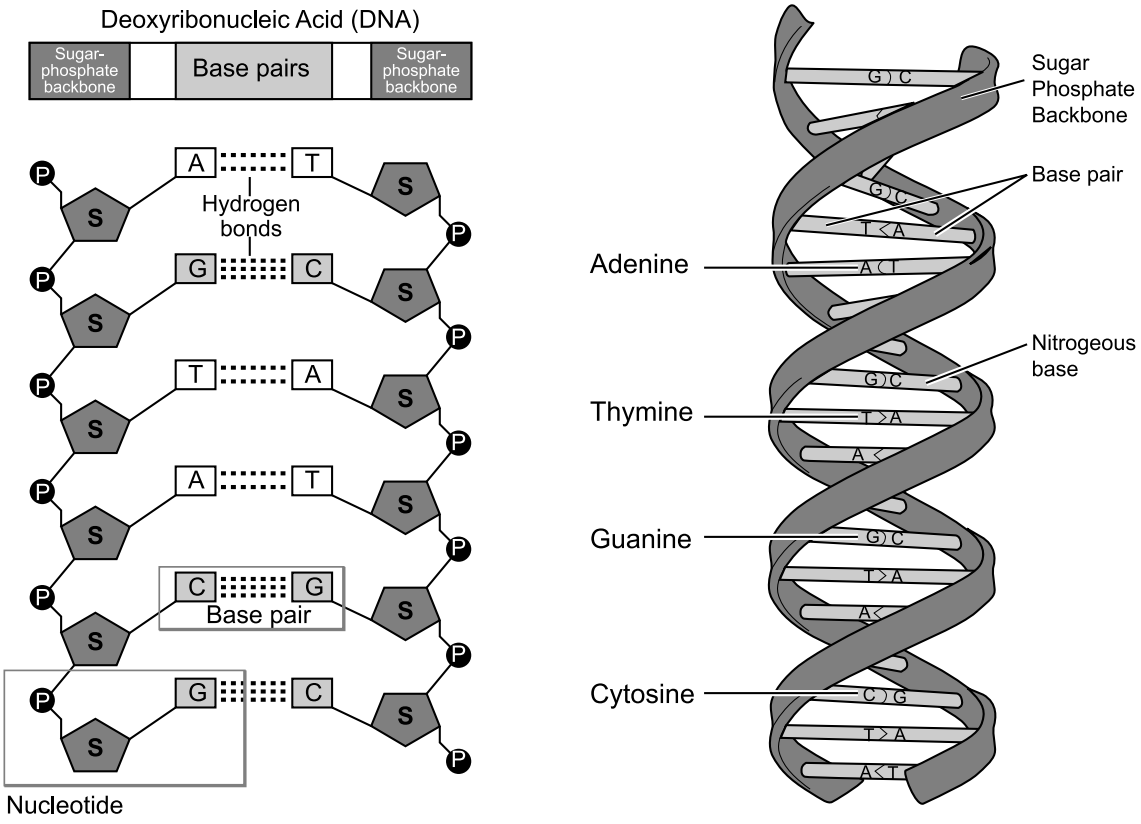

Figure 2.1: DNA Structure

Therefore, the mechanism through which DNA is used to produce proteins is crucial, and merits further elaboration.

A DNA strand contains regions of information that describe the structures of proteins, along with regions that determine the probability of mRNA/protein production by attracting or repelling ribosomes. The latter regions are called ”regulatory sequences”, while the former are called *genes*. (Note that some simpler organisms, particularly viruses, use RNA instead of DNA to encapsulate genetic material.) It is estimated that the human species has about 20,000-25,000 genes[27], made up of about $3x10^9$ base pairs. Parts of the sequence information on a gene are spliced out during the transcription process which will be described later, and do not code protein data. These parts are called *introns*, while

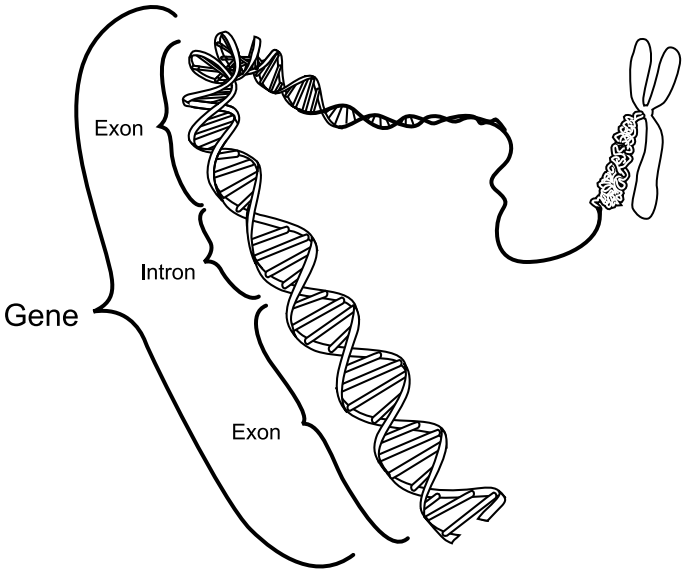

Figure 2.2: Gene Structure

the protein-encoding sections are called *exons*. Figure 2.2 illustrates the structure of a gene.

In addition to genes and regulatory sequences, DNA strands also contain a large amount of sequences that do not seem to serve a specific purpose in protein definition and production. These non-coding sequences are sometimes referred to as *junk DNA*. While there are many theories about the possible function of junk DNA, the exact functionality of these regions are not fully understood. The total DNA possessed by an organism is called the *genome* of that organism. For complex organisms, the genome is more precisely defined as the entire DNA sequence of a full set of chromosomes, which are very long strands of DNA that are found in every cell of such organisms. The general layout of a chromosome can be seen in Figure 2.3.

Each gene specifies the structure and protein of a certain type of protein with the help of regulatory regions on the genome. The process of using nucleotide sequences

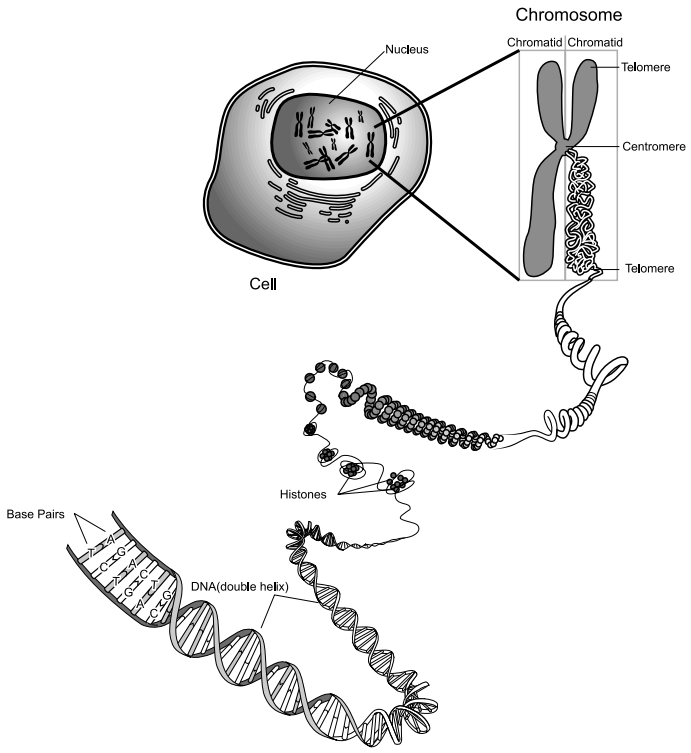

Figure 2.3: Chromosome Structure

in a gene to form proteins is called the *central dogma of molecular biology*. Figure 2.4 illustrates the process outlined in the central dogma. First, the DNA sequence replicates itself to form a copy. Then, the gene that codes the protein is converted into pre-RNA, and then a single-stranded sequence form called RNA through a process called transcription. In simpler organisms (prokaryotes), the pre-RNA step is not necessary. At the end of transcription into a form of RNA called the messenger RNA (mRNA), and the mRNA strands are released into the fluid within the cell walls (cytoplasm). The actual process of protein synthesis is accomplished by a cell organelle (small organ-like functional unit) named the ribosome, which binds to the mRNA molecules and translates the genetic information conveyed by the sequence to a protein structure. This last step of the protein synthesis process is called translation.

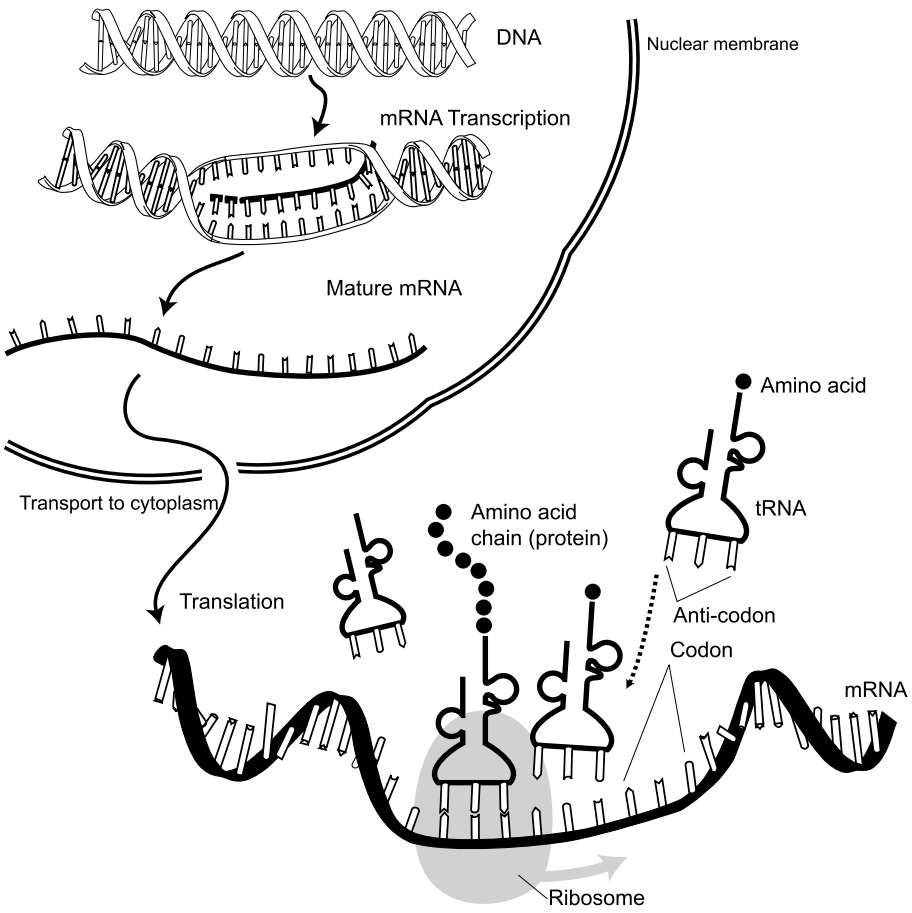

Figure 2.4: Central Dogma of Molecular Biology (Gene Expression)

The process outlined in the central dogma is also named *gene expression*, and its end results are proteins. As we mentioned earlier, proteins control and regulate vital biochemical processes; and this makes them ideal drug design targets. Many problems in bioinformatics therefore have the goal of revealing the relationships between genes and proteins they code, and ultimately determining the structural properties of proteins. Before we describe the bioinformatics application domains involved in solving these problems, we will describe the various kinds of biological data used in bioinformatics.

2.3 A Glut of Data

Probably the most important reason underlying the increasing need for computing performance in bioinformatics is the sheer size of genetic data that has been accumulating. Figure 2.5 illustrates the dramatic increase in the rate of data collection in the NCBI database. Collection and sharing of genomic data seems to have a a ”snowballing” effect in the sense that availability of genomic data from many other species facilitate comparative genomics and facilitate the discovery of new protein-coding regions in other, related species. Furthermore, the availability of large amounts of diversified biological data encourages the study of even larger, and computationally intensive problems as witnessed in the emerging field of systems biology. Even if one takes the recent advances in chip multiprocessors and proliferation of inexpensive computing clusters into account, it is evident that bioinformatics researchers might be overwhelmed by the amount of this data if the prevailing trends continue. To quote M. J. Pallen, *”Genome sequencing risks becoming expensive molecular stamp collecting without the tools to mine the data and fuel hypothesis-driven laboratory-based research.”*[34],[76].

While the practice of converting biological data into digital form has almost certainly been going on since the first years computers entered the biology and medical departments of universities, the rate of accumulation of sequence data has significantly increased in the last two decades in particular. Jones[47] links this phenomenon to two key events that have been taking place in this period: The US government’s Human Genome Project and the EST (Expressed Sequence Tag) approach to gene discovery. The Human Genome Project; an ambitious undertaking with the previously unimaginable goal

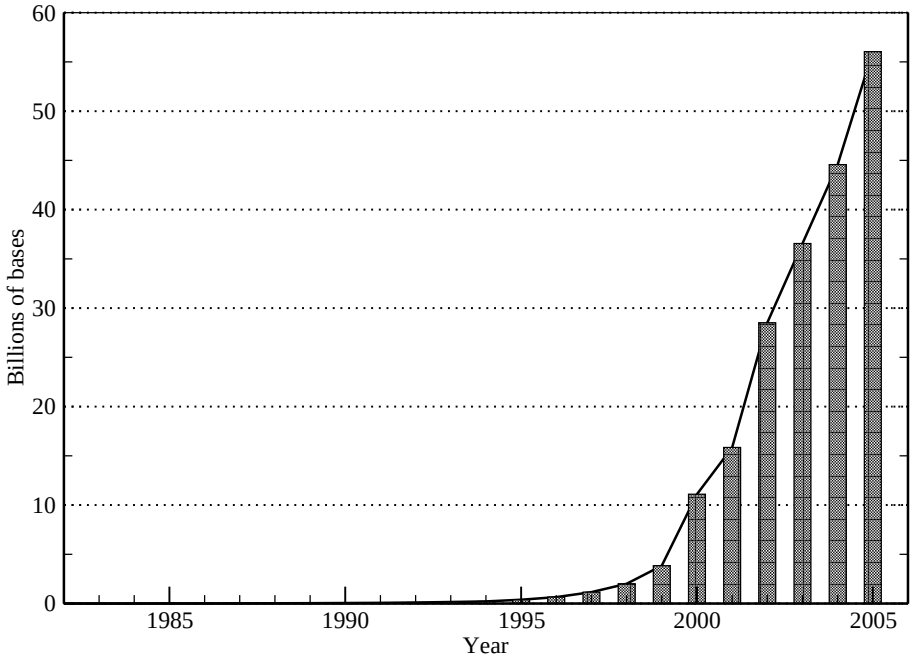

Figure 2.5: Genomic Data in the NCBI GeneBank Database

of sequencing the entire human genome; was planned in the late 1980s and initiated in 1990. In addition to generating huge amounts of data itself, the Human Genome Project contributed to other large scale genomic data collection efforts by directly or indirectly leading to many novel tools and techniques for capturing and analyzing sequence information.

The EST methodology was devised by National Institute of Health (NIH) researchers for accelerating the gene discovery process by isolating the messenger RNA molecules and using this information to isolate the entire gene later. In conjunction with the Human Genome Project effort, EST-based sequencing is generally accepted as a major driver behind the massive amount of sequence data collected in the last decade.

A basic grasp of the various data types used in bioinformatics applications is therefore clearly crucial as a background for our analysis of these applications in the following

chapters. In this section, we will briefly introduce the various kinds of data that provide the input to many bioinformatics applications, along with widely used databases holding such data, and the means through which this data is generally obtained.

2.3.1 Nucleotide and Genome Sequence Data

Nucleotide and genome sequence data form probably the largest segment of data collected by molecular biologists. Nucleotide sequence data is primarily raw DNA sequence information which might just be the sequence for a gene specifying a certain protein, or an entire genome. Such data is generally represented as chains of letters(*residues*) specifying different nucleic acids.

Nucleotide and genome sequence data together form what probably is the largest collection of biological data. Most of this data is accessible to researchers all over the world through the use of large repositories such as National Center for Biotechnology Information(NCBI) GenBank[18], European Molecular Biology Laboratory(EMBL)[25] and DNA Data Bank of Japan(DDBJ)[71] sequence databases. As of late 2005, the NCBI GenBank contains more than a hundred billion base pairs of sequence data.

Currently, most nucleotide sequence data is obtained by the EST(expressed sequence tag) method where messenger RNA molecules are isolated and cloned, and then partially sequenced in high-speed sequencers. The EST method yields large amounts of sequence data which represents the expressed portions of genes. Considered to be one of the most important factors in the growth of genomic data collection, the EST approach is described in more detail in [9].

Complete genome data is generally obtained by a technique called *shotgun sequencing*. In this method, complete sequence data for a long DNA strand is pieced together by software from shorter random segments that have been obtained by sequencing physical samples.

2.3.2 Protein Sequence Data

Since the primary structure of proteins are determined by their amino acid sequences representations, proteins can also be specified in sequence form. This representation is similar to that of nucleotide sequence data, although in the case of protein sequences each *residue* is an amino acid rather than a nucleic acid.

The central role of proteins and protein structure analysis in the drug discovery process has led to sophisticated protein sequence databases with differing levels of detail, and we will use the terminology introduced in [59] to describe these. At one end of the spectrum are the large protein sequence databases such as SwissPROT and PIR, which are primarily repositories of raw protein sequence data. In order of increasing specialization and sophistication, we observe composite protein sequence databases which store filtered and annotated non-redundant data from different databases; and secondary protein sequence databases which specialize in storing commonly conserved regions, patterns, and other information distilled from primary and composite sequence databases. Some widely used composite protein sequence databases of the first type are OWL and the NR(non-redundant) protein database from NCBI. An example for the latter group of databases is Pfam, which contains profile hidden Markov model(HMM) descriptions for

many different protein families. This kind of data is only relevant for HMM-based protein profile searching applications, one of which will be examined in detail elsewhere in this thesis.

Protein sequence data can be obtained with the use of chemical methods such as Edman degradation or mass spectroscopy methods, with the latter quickly becoming the more widely used of the two. Mass spectroscopy protein sequencing method[90] involves breaking a purified sample of the target protein into smaller units called peptides and measuring the molecular weights of the peptides using a mass spectrometer. The resulting mass spectrum can then be analyzed computationally to determine the amino acid sequence of the original protein. Data in composite and secondary protein sequence databases are obtained by processing this raw sequence data.

2.3.3 Protein Structure Data

While the amino acid sequence of a protein describes its evolutionary ancestry and possible functionality to an extent, the description of a protein only becomes complete when its *secondary and tertiary structure*, a complete specification of repeating patterns in a protein and its 3D physical structure. This data has generally been obtained by using time-consuming methods such as X-ray crystallography and NMR(nuclear magnetic resonance) spectroscopy, which explains the relatively small amount of such data in the important biomolecule 3D structure databases such as PDB[19] and MMDB[97].

Protein structure prediction, or the determination of a protein's 3D structure from its amino acid sequence, is one of the most active research areas in today's bioinformatics.

Many different approaches to this very difficult problem have been proposed, so far with limited success. *Protein folding* is a subproblem of this quest, and involves modeling the physical forces between the atoms forming the protein. Any significant future successes in protein structure prediction are likely to revolutionize the field by accelerating the drug discovery process many times.

2.3.4 Gene Expression Data

The amount of messenger RNA produced by a cell during gene expression provides information about the state of the cell and the corresponding protein levels. Since such information is valuable to understand how the cell functions and decipher the biochemical pathways for certain cell functions, researchers collect gene expression data in the form of a time series measurement of messenger RNA levels. Commonly used methods for gene expression data collection are cDNA microarrays and microarray analysis, oligonucleotide chips produced by Affymetrix, and the SAGE (Serial Analysis of Gene Expression) method. The largest gene expression data repository as of early 2006 is the NCBI GEO(Gene Expression Omnibus) [15] database.

2.4 Bioinformatics Application Domains

Collecting, analyzing and processing the diverse set of data described in the previous section calls for an even more diverse group of applications and algorithms. Some of these are more prominent than others: *Genomics*,the study of the entire genome of a given organism, is probably the most familiar bioinformatics application domain for many. Ge-

nomics itself covers a variety of different topics: While *functional genomics* involves the identification of specific functions of a gene in the genome, *comparative genomics* deals with the analysis of similarities and differences in the genomes of different species. Genomics applications are probably the most commonly used bioinformatics applications, which is not surprising considering the fact that genomic data constitutes the majority of biological data in existence. A close second is *proteomics*, which studies the entire set of proteins of an organism. While proteomics and genomics applications account for a large number of well-known bioinformatics application in use, there are many other lesser-known application domains in bioinformatics. The field of bioinformatics is still relatively new with a very diverse variety of problems and subfields; and providing a comprehensive listing of all bioinformatics applications is outside the scope of dissertation. In this section, we identify and briefly describe some of the more important bioinformatics application domains which provided the applications analyzed in the later chapters.

2.4.1 Sequence Alignment

Modern genomics relies on the paradigm of evolution which suggests that all organisms are related to each other as a result of the evolutionary process. This implies that similar proteins should exist in related species, and hence the sequences that code these proteins should also be similar. This similarity is called *homology* in molecular biology. The most important practical use of homology is in the determination and confirmation of coding regions in DNA sequences. Comparing a sequence region with regions known to encode certain proteins in other organisms can be useful for confirming whether the re-

gion might be instrumental in coding a similar protein, or not. A high degree of homology between two sequences from different organisms might also provide clues on the evolutionary links between the organisms. Sequence alignment algorithms are used to align two sequences in a way to maximize the similarity between them. In many cases, commonly used pattern matching techniques are not readily applicable to biological sequences due to the fuzzy nature of the data. To illustrate this point, we consider an alignment of the DNA sequence ”AGTAGC” with ”GAG”. An ordinary string matching algorithm will not be able to find ”GAG” within the larger string; but in case of biological sequence data we can use a gap to accommodate the non-matching character ”T”:

```
AGTAGC
-G-AG-
```

Gaps can be inserted in both the sequence that is being searched, and the sequence that is being used as the search pattern. If we were to align ”AGTAGC” with the sequence ”GCTA”, we would have:

```
AG-TAGC
-GCTA--
```

The number of matching residues(symbols) contributes positively to the similarity score; and gaps/non-matching residues contribute negatively. Sequence alignment algorithms aim to maximize the similarity score by finding the best alignment possible, which generally means that gaps should be avoided wherever possible. In the specific case of amino acid sequences, the contribution of each residue match or the penalty associated with gaps or non-matching residues is generally read from scoring matrices that were computed

with evolutionary relationships of different amino acids taken into account. These matrices are also called substitution matrices, and the most important substitution matrices in use are BLOSUM(Blocks Substitution Matrix) and PAM(Percent Accepted Mutation). These matrices are provided in several different varieties based on the degree of identity between the sequences to be aligned.

Pairwise sequence alignment algorithms described in the literature generally use a dynamic programming approach to solve what is essentially an optimization problem. The most important of these are Smith-Waterman[85] and Needleman-Wunsch[68] dynamic programming algorithms, both of which can compute optimal solutions to the pairwise alignment problem. The time and space complexity of these algorithms led researchers to design heuristic algorithms that approximate their results while allowing shorter execution times. The most widely used heuristic-based sequence alignment algorithms are BLAST[12] and FASTA[77]; both of which can align DNA, RNA or protein sequences.

2.4.2 Multiple Sequence Alignment

Just like sequence alignment methods are used to assess the extent of homology between two sequences, it is possible to extend the procedure to three or more sequences to look for signs of homology. Aligning a larger number of sequences provides more insight into how the sequences diverged through the process of evolution and which regions in the sequence were particularly prone to mutations. Multiple sequence alignment is also a useful tool in determining phylogeny; the tree-like conceptualization of evolutionary

ancestry links between different organisms or species.

Extending dynamic programming based pairwise sequence alignment algorithms to alignment of three or more sequences is not feasible due to the exponential complexity of this approach with the number of sequences[26]. Thompson et al.[92] state that multiple alignments of as few as 8 sequences with pairwise alignment techniques were not possible with the computational power available at the time of the study (1994). In order to overcome the complexity problem, a progressive alignment approach has been proposed[36] and eventually successfully implemented in several different multiple sequence alignment applications. This approach exploits the concept of homology to build a progressive multiple alignment by doing a series of pairwise sequence alignments while following the branching order of a phylogenetic tree, which shows the evolutionary ancestry relationship between the sequences. Since two sequences in the same phylogenetic tree branch are by definition closely related, the pairwise alignments get progressively more difficult as the sequences get more and more distant.

Probably the most widely used multiple sequence alignment tool is CLUSTAL W[92], which uses the progressive algorithm described above. This application generates the phylogenetic tree by calculating a distance matrix for every pair of sequences in the input set, and completes progressive alignment by using this tree as a guide. T-COFFEE[69] is another multiple sequence alignment tool which is gaining popularity; and uses a different, more computationally intensive algorithm.

2.4.3 Phylogenetic Analysis

The term *phylogeny* describes the evolutionary relationship between different species. Since such relationships can best be visualized in the form of a tree, depicting phylogeny information in a tree format facilitates further elaboration on the sequences belonging to species or biological entities thought to be related.

Most of the algorithms used for phylogenetic analysis fall within two categories depending on the type of data they use: *Distance-based* algorithms typically compute similarity metrics in the form of distances computed from pairwise alignments. In contrast, *character-based* methods directly use the multiple alignment of sequences to infer tree structures. In both cases, a wide variety of computational methods such as maximum-likelihood analysis, least-squares techniques or clustering methods can be applied to analyze the similarity data and compute the structure of the tree. A comprehensive description of various algorithms and approaches to phylogenetic analysis can be found in [75].

The most common phylogenetic analysis software in current use are PHYLIP[35] and PAUP[2], both of which have been used to generate very complex phylogenetic trees.

2.4.4 Protein Structure Prediction

As we briefly mentioned while describing protein structure data, the capability to deduce the three-dimensional physical structure of a protein from its amino acid sequence is highly desired because of its potential applications in the drug discovery process. Since the biochemical function of a protein depends on its physical shape, pharmacology researchers need to know the precise shape of the protein to be able to design a compound

that can attach to it. Currently, the only way to determine the structural layouts of proteins is through experimental methods such as NMR and X-ray crystallography.

Among all of the diverse application domains in bioinformatics, protein structure prediction probably has one of the highest (if not the highest) requirements for computing performance. In theory, inferring the physical (tertiary) structure of a protein from its amino acid sequence data should be possible through the use of molecular dynamics formulations, because the structure of the protein is dictated by the attraction and repulsion forces between the structural elements forming it. In practice, a solution by molecular modeling seems elusive, primarily because of the overwhelming computational cost and inaccuracies introduced by the experimental processes needed to determine the parameters needed for the computation. As a result, current research in this field has focused on statistical and empirical methods which exploit the existing repository of protein structure data.

Since the emerging field of protein structure prediction is still very active and many important questions remain answered, it is somewhat difficult to name applications in wide use. An example application THREADER[46] which evaluates the compatibility of known protein folds and amino acid sequences. Rost et al.[83] describe the important issues in protein structure prediction while providing a good introduction to this application domain.

2.4.5 Systems Biology: The Holy Grail

While the insights provided by genomics, proteomics and functional genomics have allowed scientists to understand the origins of life better and formulate drugs for diseases, they do not afford us a complete understanding of the complex events that take place in even the simplest organisms. An emerging paradigm called systems biology aims to combine the numerous computational biology techniques and data in order to model entire biological systems such as tissues, organs, even complete life forms. The ambitious goal of being able to model extremely complex biological systems, if realized, could allow scientists with a very useful tool to observe important biochemical pathways *in silico* and even test the effects of new medications using computer simulations.

Since systems biology is a very recent concept, the kind of applications and data formats it will require are not fully determined yet. The breadth of the effort implies that algorithms, workloads and data formats from all domains of bioinformatics are likely be utilized. A good and concise introduction to the concepts and challenges of systems biology is presented by Morel et al. in [64]. Finkelstein et al.[37] present a useful discussion of the computational requirements of the emerging field of systems biology, centering on the modeling challenge.

Needless to say, computational requirements of systems biology applications will be enormous; and the field is already being recognized as a ”grand challenge” in high performance computing. At the time this chapter was being written, Freddolino et al. have published the results of the first complete simulation of an entire life form in [39]. They used the NAMD molecular dynamics framework[80] to simulate the modeled virus

for 13ns. of simulated time. Even though the simulated life form(the satellite tobacco mosaic virus) was a very simple and primitive organism containing a total of two proteins, their full simulation ran on a 256-node (512 processors) Intel Itanium 2-based SGI Altix NUMA SMP system at a speed of 1.1ns of simulated time per day.

2.5 Bioinformatics and The Drug Discovery Process

In order to maintain growth and meet their targets, drug companies need to develop at least several new drugs with high sales potential every year. An important reason behind this pressure on pharmaceutical companies is the relatively short patent protection periods for drugs. Once a company loses patent protection on one of its flagship drugs, revenues from that product fall precipitously as competitors rush to produce generic versions. Faced with the difficulty of coming up with suitable candidates for such drugs using traditional techniques, large pharmaceutical companies started exploring the use of bioinformatics in drug design during the mid-1990's[47].

Traditional drug discovery heavily relied on animal models of diseases and chemicals whose therapeutic values have been determined to some extent. In contrast, bioinformatics-centered drug discovery starts with the identification of genes associated with a certain disease or desired drug response. The detection of disease-related genes can be accomplished through the use of microarrays that detect the presence of related mRNA. Using comparative genomics techniques, similar sequences in other organisms can be identified and a human protein structure can be modeled. Armed with this information, researchers can then target this protein and design a compound to bind to this molecular target.

Commonly referred to as the *rational drug design*, this approach to drug discovery requires the use of many different tools in the bioinformaticists' arsenal[59, 8]. Genomics applications such as sequence alignment are used to find homology between human genes and their counterparts in other species. Protein profile searching applications might be used to find similar patterns in human and animal proteins, and protein structure prediction techniques play an important role in finding the chemicals to bind to the physical structures of target proteins. The interplay of different applications and algorithms in the rational drug discovery process suggests that the performance of these diverse applications is crucial for success in rapid and efficient drug discovery. The obvious potential scientific and financial rewards of rational drug discovery are important driving factors behind the research activity in bioinformatics.

Chapter 3

Workload Characterization of Bioinformatics Applications

3.1 Overview

While the design and analysis of faster algorithms for bioinformatics applications is a very active field of research, very little had been published in the literature on general performance characteristics of these applications and the implications on system or processor design at the time of the start of our studies for this thesis. Most of the published work in this field seemed to have focused on incremental improvements to bioinformatics application suites or certain algorithms. One reason behind the apparent disconnect between computational biology and computer architecture communities could be the lack of a standard benchmark suite of bioinformatics applications.

We saw a clear need for such a set of well-defined benchmark applications drawn from bioinformatics codes in common use, which will be an important step towards motivating further research on the characteristics of such applications and their implication on computer systems engineering. We studied important application domains in bioinformatics and identified most widely used bioinformatics applications for further analysis. This chapter describes the results of our efforts: BioBench, a benchmark suite of bioinformatics applications chosen to reflect the diversity of bioinformatics codes in common use. The applications in BioBench and the reference data sets were selected with input from the bioinformatics community, and we expect BioBench to evolve in response to

future developments and comments from both bioinformatics and computer architecture communities. The initial BioBench suite aimed to provide tools to evaluate bioinformatics applications on uniprocessor systems, a parallel version is part of our future plans for BioBench.

In addition to providing a benchmark for evaluating the performance of computer systems running common bioinformatics applications, a secondary goal of the BioBench suite is to establish bioinformatics applications as a distinctly different class of applications than the commonly accepted framework of scientific applications. In contrast to these scientific applications which typically are floating-point intensive, many bioinformatics applications operate with textual representations of biological sequence data. The straightforward encoding of this data can mean that many bioinformatics codes are primarily fast string search or pattern matching applications; and we have reason to expect distinctly different execution behavior for these benchmarks than traditional scientific application benchmarks, particularly with respect to the importance of floating point versus integer operations and branch behavior. In the benchmark characterization part of our work, we obtain basic execution characteristics for the applications present in the BioBench suite, and compare these characteristics to those of SPEC 2000 benchmarks to test the validity of our expectations.

The rest of this chapter is organized as follows: Section 2 describes major bioinformatics application domains represented in BioBench, and describes the applications. Section 3 describes our experimental methodology and tools used to obtain performance data. Section 4 presents this data, and compares the characteristics of BioBench applications to those of SPEC 2000 benchmarks. We present some of the related work in Section

5, and finally, we present some concluding remarks in Section 6.

3.2 BioBench Suite Applications

An important goal of the BioBench effort was to define a representative set of bioinformatics application domains. We first identified several important application classes and selected commonly used applications from these classes. While a diverse set of benchmark applications was desirable, we limited the scope of this initial release of BioBench to relatively mature application classes that found widespread usage in academia and industry. In addition to widespread use, another important criterion in choosing benchmarks was the availability of source code for use in our studies, and in some cases a relatively less known application suite had to be chosen instead of a popular commercial suite.

Equally important was the selection of input data that is representative of real-world computational biology problems. Problem sizes were determined in collaboration with members of the bioinformatics community, and our execution-based methodology allowed us to use complete copies of major protein and DNA databases instead of smaller data sets which would not be representative of real-world problems and could have skewed the results. As an example, the BLAST workload in BioBench was evaluated using NCBI's NT database, containing 11GB of data that represented all DNA sequences discovered to date.

We recognize that bioinformatics is a very diverse field, and the initial version of BioBench does not cover some important application domains like microarray analysis,

protein structure prediction, protein docking and spectrometry. In its initial version, the choice of BioBench applications reflects an emphasis on mature genomics tools. (Future versions will address a much wider variety of bioinformatics application domains. As new application domains emerge, we plan to update BioBench with new benchmarks.)

The application classes and the individual BioBench benchmarks selected to represent them are listed below.

3.2.1 Bioinformatics Application Domains Represented in Biobench

Sequence Similarity Searching

Sequence similarity searching applications are typically used to identify similarities between DNA or protein sequences, or to search for certain subsequences in large sequence databases. The similarity between two sequences (or the lack of it) can often reveal important clues about structural or functional relationships between them, and in some cases can provide important clues about common evolutionary roots of organisms. As described in the introduction of this thesis, the nature of the sequence data necessitates a more complex search mechanism than simple text search, and exact algorithms like the Smith-Waterman algorithm has been developed for this purpose. The complexity of the Smith-Waterman algorithm in turn has led to research on approximation-based methodologies for sequence similarity searching, and the most commonly used applications in this field are both based on algorithms that approximate Smith-Waterman. BioBench contains programs from both BLAST [12] and FASTA [77], the two most widely used suites for sequence similarity searching.

- **BLAST**: The most commonly used sequence searching application is represented by two programs, BLASTN and BLASTP, in the BioBench suite. These programs are used for DNA and protein sequence searching, respectively. We used the freely available version 1.3 of the BLAST suite. The DNA and protein databases used were NCBI's NT (11GB) and NR (945MB) databases containing the full set of non-redundant of DNA and protein sequences submitted to NCBI.

- **FASTA**: BioBench includes the main search utility from University of Virginia's FASTA suite v3.4t21, the other important sequence searching suite [77]. FASTA is generally accepted to be slower than BLAST, but it is the preferred application in some cases due to its higher sensitivity and better tolerance to gaps. Just like BLAST, FASTA contains applications for searching protein and DNA sequences. To reflect the difference between protein and nucleotide (DNA) searches, our test cases use the FASTA application for searching against a DNA database and a protein database with suitable search sequences. The DNA database used in our study is a daily update file to the NCBI GenBank data repository (190MB), and the protein database used is the entire SwissPROT protein database(70MB) at the date of this study.

Phylogenetic Analysis

Phylogenetic analysis aims to discover how a group of related protein sequences were derived from common origins during the process of evolution. This information is frequently displayed as a hierarchical diagram called a phylogenetic tree. The discovery

and visualization of such relationships between proteins offers important clues on how certain traits were passed from species to species, and the results from phylogenetic analysis are commonly used to guide further analysis on proteins deemed to share common ancestry.

- **PHYLIP**: To represent phylogenetic analysis applications, we chose a benchmark from the PHYLIP suite [35], version 3.5c. PHYLIP is the most widely used phylogenetic analysis package, and contains several programs to conduct different types of phylogenetic analysis. The PHYLIP application chosen for inclusion in BioBench is PROTPARS, a protein parsimony computation application.

Multiple Sequence Alignment

Multiple sequence alignment is the process of aligning more than two sequences to find regions of similarity. This kind of analysis is used to have a deeper understanding of similarity patterns that might suggest common origins between the proteins they code.

- **CLUSTAL W**: For our representative multiple-alignment benchmark, we chose the CLUSTAL W multiple sequence alignment application. CLUSTAL W [92] builds on the CLUSTAL package described in [43], and is currently the most commonly used multiple sequence alignment application.

Protein Sequence Profi le Searching

The process of evolution introduces a degree of randomness in the amino acid sequences which define proteins, causing additions or deletions of amino acids over time.

This property of protein sequences necessitates the use of a fuzzy search mechanism. When an evolutionary diverse set of proteins are under investigation to find remotely related (and therefore functionally similar) proteins, searching a sequence database for a profile of a sequence family (a common signature of the family) can be more effective than searching the same database for individual sequences. This analysis approach is called sequence profile searching, or protein motif searching. The most common method for this type of search involves the use of hidden Markov models, a probabilistic approach commonly used in fields like speech recognition.

- **HMMER**: We selected the most commonly used protein sequence profile searching program,HMMER[31], to represent this class of applications in BioBench. HMMER uses a hidden Markov model(HMM) based approach to conduct searches of protein motifs against protein sequence databases, or single protein sequences against protein motif databases. We used the hmmsearch application from the HMMER v2.3 to search the entire SwissPROT protein sequence database against the consensus of a small selection of protein sequences.

Genome-level Alignment

Genome-level alignment algorithms and tools are used to align complete genomes of related species. Due to the sheer number of nucleotides in a complete genome, multi-sequence alignment algorithms and tools (which are more geared toward aligning single proteins or simple nucleotide sequences) can not be used effectively for this task. Genome-level alignment tools employ algorithms specifically developed for the purpose

of pairwise alignment of very large nucleotide sequences.

- **MUMMER**: MUMMER[29] is a genome-level alignment tool that uses suffix trees to assemble complete genomes. We chose MUMMER v3.14 for inclusion in BioBench.

Sequence Assembly

Sequence assembly tools are used to generate sequence data from many small overlapping partial sequences obtained by DNA sequencing hardware. Also called shotgun sequencing, sequence assembly is a crucial step in obtaining sequence data from physical DNA sequences.

- **TIGR**: The class of sequence assembly applications is represented by the TIGR Assembler[91] suite in BioBench. The version we used in BioBench was TIGR Assembler v2.

3.3 Methodology and Tools

Many of the bioinformatics applications we selected, such as sequence similarity searching and multiple alignment, are typically used in conjunction with very large databases, resulting in large execution times that are impractical for a simulator-based study. In order to meet our goal of collecting data on the entire execution of bioinformatics applications with meaningful input sizes, we decided that the most suitable experimental methodology was to use the hardware performance counters built into modern microprocessors rather than a processor simulator.

Many modern microprocessors include special-purpose counters that can be used

to count occurrences of different events and registers to access these counters. Among the many different events that can be counted are cache misses, branch mispredictions, and others that are useful measures of application performance. A particular drawback of hardware performance counters is their limited number: there were only 2 on the Intel Pentium 3 CPU used in our study. One workaround for this limitation is multiplexing, which uses time-sharing to use the counters to measure different events at different time slices, and extrapolates the result. For long-running applications (which is typical for bioinformatics application workloads), the multiplexing method yields reasonably accurate measurements [62]. We used the PAPI hardware performance counter access library [21] that uses the *perfctr* Linux kernel patch for counter multiplexing. To facilitate data collection and analysis, we used the PerfSuite [3] utilities.

Using these software to utilize CPU performance counters, we were able to run unmodified BioBench applications with large input sizes characteristic of their typical use. We used a commodity workstation based on an Intel Pentium 3 CPU running Linux kernel 2.4.22; and PAPI v3.0.(At the time we ran our experiments(early 2004), a Pentium 3 based machine was the latest available to us) All BioBench programs were compiled using *gcc* version 2.95 on the same computer system used for data collection, at the *-O4* optimization level. To collect some low-level hardware performance counter data not collected by PerfSuite/PAPI, we also used the *brink/abyss* [89] toolset.

To provide a comparison to the SPEC benchmark suite, applications from the SPEC 2000 suite were also compiled using the same compiler and system using the default parameters. We collected execution characteristics using complete reference data input sets from the SPEC distribution, to be used for comparison against the BioBench benchmark

Benchmark	Description	Instruction Count
blastn	DNA sequence searching	215,131,057,029
blastp	Protein sequence searching	514,628,929,894
clustalw	Multiple sequence alignment	2,150,900,967,391
fasta_dna	DNA sequence searching	1,001,512,078,272
fasta_prot	Protein sequence searching	1,149,078,024,873
hmmer	Sequence profile searching	1,573,753,830,214
mummer	Genome-level alignment	106,703,486,044
protpars	Phylogenetic analysis	1,730,029,486,107
tigr	Sequence assembly	862,484,000,000

Table 3.1: BioBench Benchmarks and Workload Instruction Counts

Processor	Intel Pentium III
Clock Speed	700 MHz
Main Memory	512MB
L1 data cache	16KB, 4-way set assoc.
L1 instr. cache	16KB, 4-way set assoc.
L2 cache	256KB, 8-way set assoc.
Cache Line Size	32B

Table 3.2: Parameters of the system used in the study

applications.

Our execution-based methodology allowed us to collect precise performance characteristics on a real commodity processor for entire workloads that took up to 2.1 trillion instructions. The number of instructions for each benchmark in the BioBench suite are presented in Table 3.1. Some pertinent parameters of the Intel Pentium 3-based system used for our study are given in Table 3.2.

3.4 Benchmark Characteristics

To characterize the BioBench suite, we collected detailed data on instruction profiles, basic block lengths, IPC, L1 and L2 data cache miss rates, and branch prediction accuracy. The same set of data were collected for the SPEC 2000 benchmarks for com-

parison.

3.4.1 Instruction Mix

In the first phase of the evaluation, hardware performance counters were used to provide a count of instructions belonging to different major instruction classes in the x86 architecture. Instruction profiles for BioBench applications are given in Figure 3.1. For comparison, the average instruction class percentages for SPEC INT and FP benchmarks are also shown. The instruction class profiles of BioBench applications reveal several interesting points: First, we observe that the floating point operation content of almost all BioBench applications are negligible. This finding reflects the intuition that bioinformatics(particularly genomics) applications are inherently different from mainstream scientific codes due to the representation of the sequence data they operate on. None of the BioBench workloads contained a floating point instruction content of more than 0.09 % of all instructions executed. While operating on primarily string data, most of the benchmarks do rely on some floating point computation for calculating statistics and likelihood values as part of their main algorithms, but this does not seem to constitute a significant part of the overall instruction count.

The average share of load instructions in BioBench applications has a marked difference from that of SPEC integer benchmarks, and these instructions constitute a larger portion of the instruction mix in BioBench than in both classes of SPEC benchmarks. This implies that the amount of computation per datum is relatively small, which is a typical character of many search algorithms(Many of the BioBench components search through

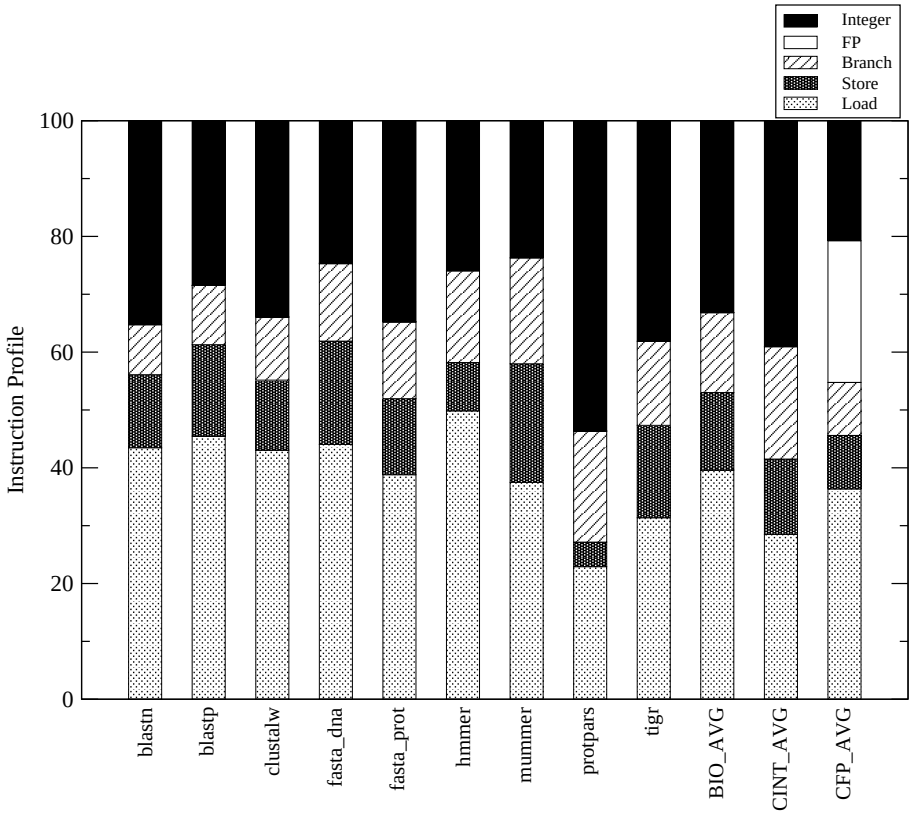

Figure 3.1: Instruction profiles for all BioBench benchmarks

large input files and databases.). The BioBench component with the lowest share of loads, *protpars*, was also the one benchmark with the smallest input file size in the benchmarks. (It is the second longest-running workload in BioBench, however.) *Protpars* essentially is less of a database search application than many of the BioBench components are, since its main function computes a tree-like hierarchy for related species using relatively shorter sections of sequences. This benchmark also differed from the rest of BioBench components with its larger share of integer ALU instructions, these instructions accounting for more than half of the instruction count. Similarly high share of load instructions was observed in one other non-search component, namely *mummer* which was found to be highly memory-bound with its dependence on very large suffix-tree data structures cre-

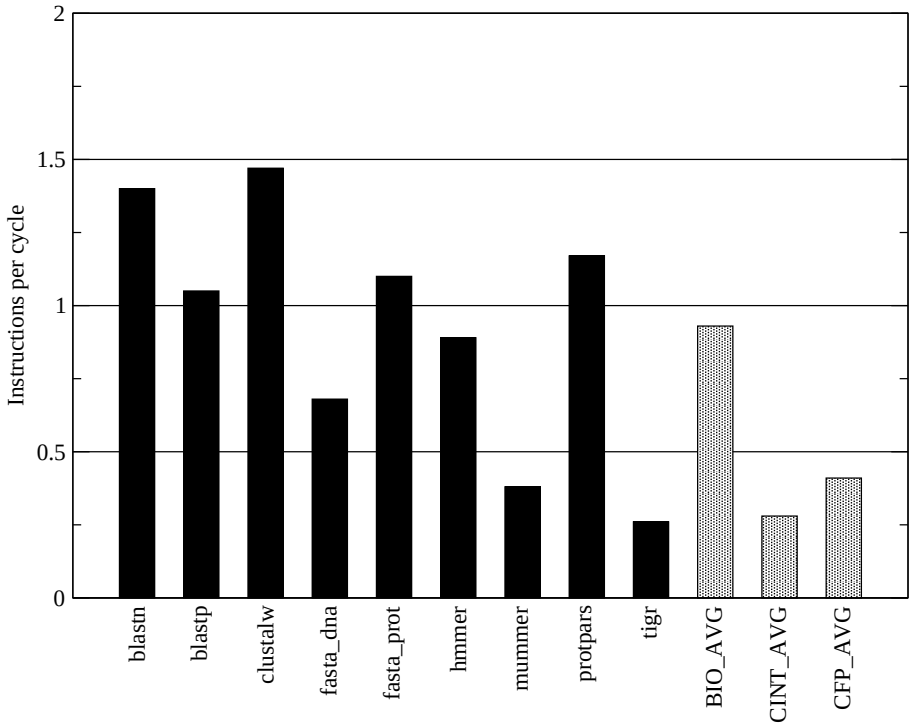

Figure 3.2: IPC values for all BioBench benchmarks and SPEC averages

ated in memory. The higher share of load/store operations in BioBench suggests that bioinformatics applications might benefit from future architectures with higher memory bandwidth and prefetching.

3.4.2 ILP(Instruction Level Parallelism)

Figure 3.2 shows the IPC values for the applications in the BioBench suite. The somewhat higher average IPC of the BioBench benchmarks hint at higher levels of ILP (instruction-level parallelism) in the BioBench applications than the SPEC INT and FP benchmarks. There seems to be a marked difference between the ILP levels in the BioBench suite and the SPEC INT suite, to the degree that only one BioBench application(*tigr*) has an IPC level lower than the SPEC INT average. This finding is encouraging,

and along with our earlier finding of almost negligible floating point content in BioBench, suggests that the basic algorithms underlying these bioinformatics applications will benefit from either wider superscalars of the future, or future chip multiprocessors with highly optimized fast integer execution units. In light of the recent microarchitectural trends and the increasing popularity of chip multiprocessors(CMPs), the finding that bioinformatics applications exhibit a fairly good degree of fine-grained instruction-level parallelism in addition to relatively easily exploited coarse-grained parallelism bodes very well for these workloads. However, the average level of ILP still seems limited, which suggests that we need to look elsewhere(thread-level and data parallelism) for even higher performance gains for these applications.

The IPC values display a great deal of variability among benchmarks. *blastn* and *clustalw* have the highest ILP levels among the benchmarks(It should come as no surprise that these benchmarks also display higher degrees of branch prediction accuracy and higher basic block lengths.). In turn, *mummer* and *tigr* have the lowest ILP. This dismal performance seems to be related to the memory access characteristics of these benchmarks: *mummer* in general is a highly memory-bound application which also seems to suffer from high L1 and L2 data cache miss rates; and *tigr* has the highest L1 data cache miss rates among all BioBench benchmarks. While we anticipated to observe high levels of ILP in bioinformatics codes due to the often mentioned “embarassingly parallel” nature of these programs, we did not expect to see this level of difference between BioBench and SPEC suite. The considerable variation in the IPC values for the individual applications in BioBench is noteworthy, and our future work on BioBench will include a detailed analysis of performance differences between applications that are very similar in function

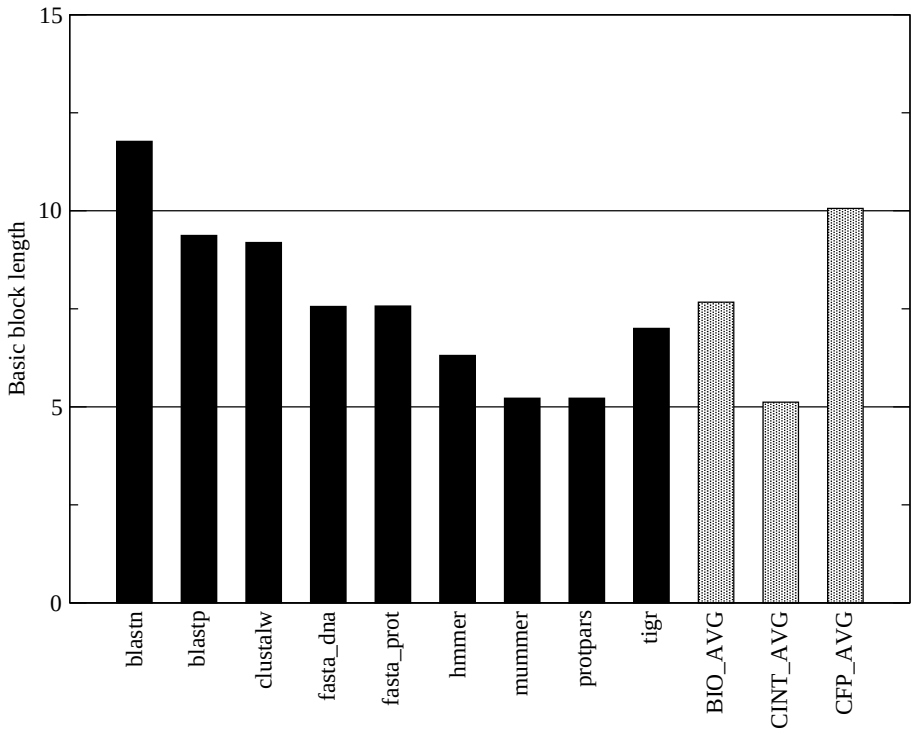

Figure 3.3: Basic block length for all BioBench benchmarks and SPEC averages

and usage, a clear example being *blastn* and *fasta_dna* which execute essentially the same kind of search using two different algorithms.

3.4.3 Basic Block Length

The basic block length for BioBench applications is shown in Figure 3.3. The average basic block length of the BioBench suite suggests that roughly one in every eight instructions is a branch. Sequence similarity search and multiple alignment applications have somewhat longer basic block lengths than the rest of the benchmarks, suggesting a more regular control flow. On average, BioBench applications have a basic block length that lies roughly between those of the SPEC INT and SPEC FP averages; all individual BioBench benchmarks having higher basic block lengths than the SPEC INT average.

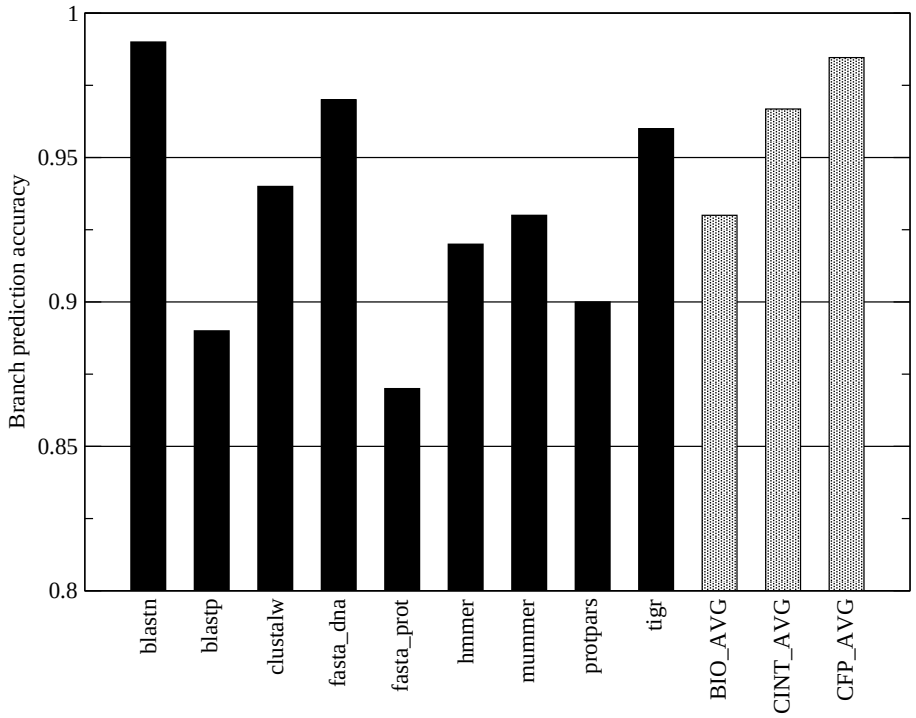

Figure 3.4: Branch prediction accuracy for all BioBench benchmarks and SPEC averages

The higher basic block length for applications in BioBench support the finding that bioinformatics applications are closer to scientific workloads than integer workloads in terms of the distribution of control transfer instructions.

3.4.4 Branch Prediction Accuracy

3.4.5 L1 and L2 Data Cache Miss Rates

Figure 3.4 shows the branch prediction accuracy for the benchmarks. While the branch prediction accuracy for BioBench benchmarks is somewhat lower than that for SPEC benchmarks, the difference is not significant considering the very high prediction accuracy available with modern branch predictors. In general, the branch prediction mechanisms in modern microprocessors seem to be working well for bioinformatics

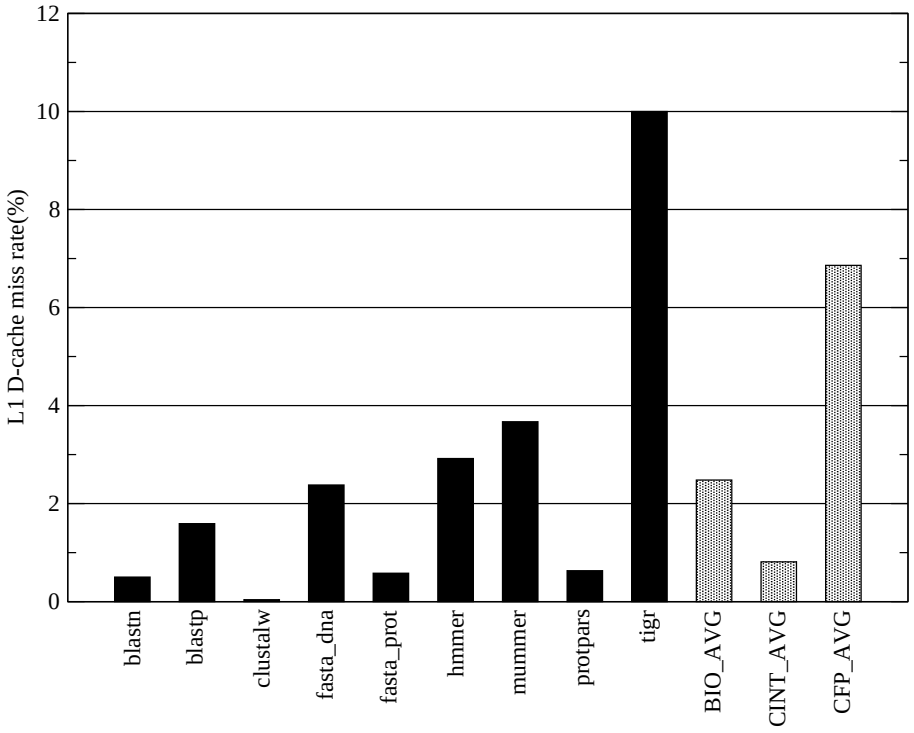

Figure 3.5: L1 data cache miss rate for all BioBench benchmarks and SPEC averages

applications.

An interesting observation in the predictability of branches in protein and DNA searching applications: the branches in *blastp* and *fasta_prot* seem to be much less predictable than their nucleotide(DNA) counterparts *blastn* and *fasta_dna*. These two benchmarks are the only entries in BioBench with less than 90 percent branch prediction accuracy. This might be linked to the higher number of possible symbols in protein sequences(20 vs. 4). In fact, the two protein similarity search applications have the least predictable branches in the suite.

Considering the size of the genetic databases that most bioinformatics applications use, the memory subsystem behavior of these workloads will be a substantial determinant of their overall performance and a detailed study of this is presented in another chapter of

this dissertation. As a part of our study, we also looked at L1 and L2 data cache miss rates of BioBench applications that were obtained using the processor performance counters (BioBench applications display negligible instruction cache miss rates, therefore I-cache results are not included in this study). L1 and L2 data cache miss rates are shown in Figures 3.5 and 3.6, respectively, and highlight differences in memory usage patterns of different BioBench components. The genome-level alignment program *mummer* and the sequence assembly program *tigr* have higher L1 data cache miss rates than the rest of the applications in BioBench, a characteristic mirrorred by their L2 data cache miss behaviors. These two applications had very high levels of memory utilization that eventually led us to scale the problem size for *mummer* down to be able to run it to completion on our test system with 512MB of main memory. In contrast, the multiple alignment component *clustalw* displayed very low L1 and L2 data cache miss rates. The component with largest duration of execution in our studies, *clustalw* displayed high IPC and fairly high average basic block length in addition to its low memory footprint. To our knowledge *clustalw* is one of the few commonly-used computational biology applications that had not been implemented in hardware before, and we believe its characteristics warrant a closer look at this benchmark as part of our future work. *fasta_prot* also displays a similarly low L1 and L2 miss rate, suggesting high data locality.

3.4.6 Execution Profiles

Many performance characterization studies in the literature treat the workload as a black box, and do not attempt to provide a view of how and where the workload spends

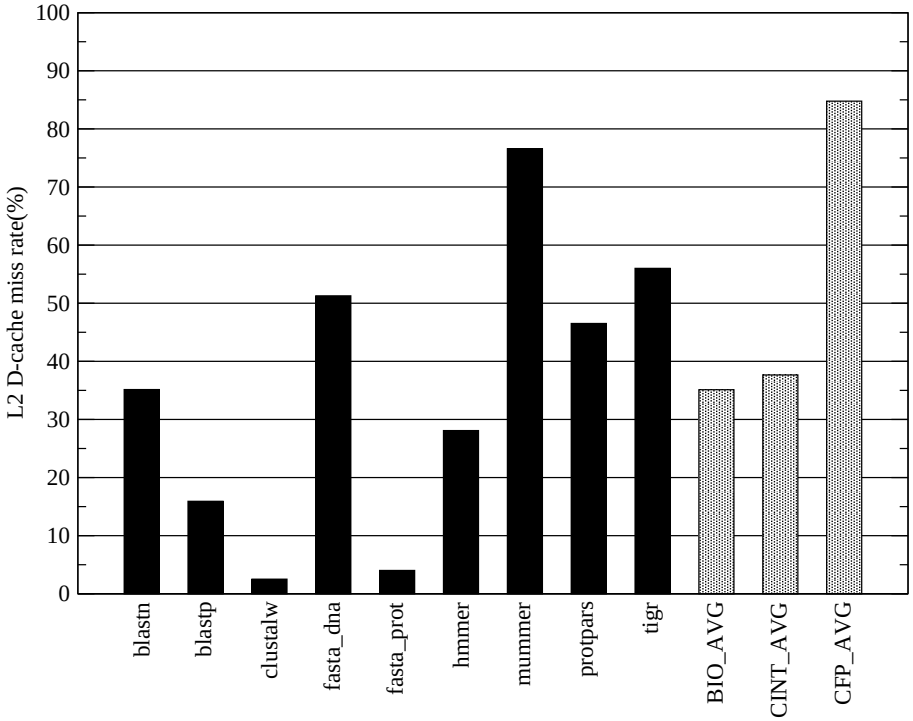

Figure 3.6: L2 data cache miss rate for all BioBench benchmarks and SPEC averages

its execution time. Most workloads have a small number of functions where most of the execution time is spent; and studying these (often short) parts of the code can yield useful insights for better understanding of its performance characteristics potential for future optimizations. We believed that a study of the breakdown of total execution time for BioBench benchmarks would be helpful for determining where optimization efforts should be focused. To this end, we used *gprof* [41] to profile the execution of BioBench applications on an 2.8GHz Intel Pentium 4 based Linux workstation with 2GB of RAM. All the workloads were compiled by gcc using the same settings used in the characterization study, and the same reference inputs were used. After a complete run of each benchmark, we tabulated the top 5 functions with respect to percentage of execution time. For each of the nine benchmarks, the total percentage of execution time spent in the top

BLASTN

Rank	Function	% of Run Time
1	BlastNtWordFinder	84.85
2	BlastNtWordExtend	10.72
3	ALIGN_packed_nucl	2.09
4	readdb_get_sequence	1.49
5	readdb_get_sequence_ex	0.10

BLASTP

Rank	Function	% of Run Time
1	BlastWordFinder_mh_contig	72.08
2	SEMI_G_ALIGN_EX	13.73
3	BlastWordExtend_prelim	11.18
4	BLASTPerformFinalSearch	0.42
5	readdb_get_sequence	0.32

FASTA_DNA

Rank	Function	% of Run Time
1	bg_align	26.27
2	FLOCAL_ALIGN	23.84
3	LOCAL_ALIGN	23.79
4	do_fasta	17.94
5	discons	2.34

FASTA_PROT

Rank	Function	% of Run Time
1	FLOCAL_ALIGN	43.26
2	do_fasta	25.29
3	savemax	3.94
4	spam	3.91
5	vfprintf	2.93

CLUSTALW

Rank	Function	% of Run Time
1	diff	51.16
2	forward_pass	23.97
3	backward_pass	21.61
4	pdiff	1.83
5	open_penalty2	0.21

HMMER

Rank	Function	% of Run Time
1	P7Viterbi	98.21
2	P7ViterbiTrace	0.69
3	index	0.31
4	_int_malloc	0.11
5	SymbolIndex	0.08

MUMMER

Rank	Function	% of Run Time
1	scanprefixfromnodestree	40.89
2	rescanstree	32.26
3	write	6.89
4	rescan	4.66
5	scanprefix	4.65

PROTPARS

Rank	Function	% of Run Time
1	fillin	98.77
2	evaluate	0.85
3	preorder	0.10
4	savetree	0.10
5	savetraverse	0.07

TIGR

Rank	Function	% of Run Time
1	find_align	42.94
2	computePairRecords	29.83
3	buildHash	8.80
4	best_match_compare	4.86
5	reset_coords	3.59

Figure 3.7: Top 5 functions according to percentage of execution time in BioBench workloads

five functions range from 94.6 percent(BLASTN) to 99.2 percent(HMMER), therefore just listing the top five functions gives a fairly thorough picture of the breakdown of execution time. Figure 3.7 displays this data in a table.

HMMER exhibits a very interesting profile, with one function dominating the entire run of the program. The *P7Viterbi()* function, which takes up to 98.21 percent of the execution time in our experiment, implements the *Viterbi algorithm*, the algorithm used in *hmmer* search application for computing the overall probability of a path through the hidden Markov model representation of a protein profile. Clearly, the impact of *P7Viterbi()* on the performance of the *hmmer* search workload is so high that any effective optimization of this function can boost the performance of this workload significantly. This property of HMM profile searching has been used to accelerate this workload on several different platforms: Lindahl[**?**] used AltiVec SIMD extensions to accelerate the *P7Viterbi()* function to significantly increase the performance of *hmmer* on IBM and Motorola (Freescale) PowerPC platform. As we will explain in Chapter 5, the almost total dependence of *hmmer* performance on one relatively compact function was one of the most important reasons for choosing this workload as our case study for accelerating bioinformatics workloads on the Cell Broadband Engine multiprocessor.

The execution time of *clustalw* is dominated by a few functions: Only three different functions (*diff(),forward_pass()* and *backward_pass*) account for 96.74 percent of the execution time. A close look at the benchmark source code and the related work of Thompson et al.[92] reveals that the algorithm used in *clustalw* consists of three stages: Calculation of the distance matrix, guide tree calculation and progressive alignment. The *diff()* function, which takes up 51.16% of the total execution time, implements a memory-

efficient pairwise alignment algorithm (described by Myers and Miller in [67]) for the progressive alignment stage. It is mentioned in [92] that the algorithm sacrifices computation speed for memory efficiency, which suggests it might be possible to replace it with a faster algorithm on systems with sufficiently large memory for large alignments. The other two functions are used in the pairwise alignment stage.

Protpars seems to spend almost all of its execution time (98.77 percent) in a single function named *fillin*; which is a relatively straightforward and short function that operates on a tree structure and counts the changes among the subparts of the tree. textitProtpars was one of the more time-consuming applications we studied; and the relatively high L1 and L2 cache miss rates of this benchmark suggests that frequent cache misses caused by the random, pointer-chasing nature of memory accesses in this function is probably the most important factor responsible for its long run time and low performance.

FASTA uses a modified Smith-Waterman algorithm which works on a band of residues, which is referred to as a "banded Smith-Waterman" algorithm. Most of the execution time of *Fasta_dna* are spent in the band boundary detection and band alignment functions *LOCAL_ALIGN* and *bg_align* functions, as well as the banded Smith-Waterman algorithm implementation itself in the form of *FLOCAL_ALIGN*. For protein sequence searching against a protein database, FASTA uses a full Smith-Waterman algoritm, and the *FLOCAL_ALIGN* function (executed with no band restrictions) takes up to 43.26 percent of the execution time for *Fasta_prot*. The performance characteristics of the Smith-Waterman algorithm have been studied extensively, and any optimization of this function could be applicable to accelerate FASTA in both protein and DNA searching modes.

A majority of the execution times of both BLAST applications are spent in the

BlastNtWordFinder and *BlastWordFinder_mh_contig*(for BLASTP) functions which implement the heuristic algorithm used to search words which will yield high scores after alignment and scoring using the substitution matrix. BLAST is generally accepted to be I/O bound, and previous attempts in the literature to improve its performance yielded only limited results (with the exception of FPGA-based hardware implementations such as), suggesting that these functions remain difficult targets for acceleration.

A close look at the most-time consuming functions in the *mummer* and *tigr* benchmarks reveal a similar structure where the main algorithm is divided into two major functions which take up to more than 80 percent of the running time(In both of these benchmarks, the rest of the execution time is spent in a large number of other functions; which do not show up in the top 5 functions in Figure 3.7.). In the case of *mummer*, these two functions are used to create and scan the suffix tree data structures. This application has been continually improved in recent years, and most of the improvements seem to have centered on decreasing the large memory footprint of the algorithms implemented in these functions.

3.5 Related Work

Many examples of application domain-specific benchmark suites have been proposed and some were widely accepted, following the example of the SPEC [87] suite for evaluation of integer and floating-point performance of computer systems. Among the most notable are the TPC benchmarks [93] for database/transaction processing, and more recently benchmark suites like MediaBench [56] or CommBench [99].

To the best of our knowledge, a comprehensive set of bioinformatics benchmarks has not been compiled and studied prior to our study. In contrast, studies on performance of individual algorithms or tools were abundant in literature, and most of the published work on performance studies of bioinformatics workloads involved either performance optimization of established algorithms, or analysis of the performance of such algorithms on parallel systems. Yap et al. [102] present a detailed study of parallel sequence searching. Catalyurek et al. [23] analyze performance of specific applications on a centralized-server, multi-client environment. While we could not find any comprehensive academic study of multiple bioinformatics workloads that predated the first publication of our work, we noticed at least one publication on the subject from the industry: The SGI Bioinformatics Performance Report [4] includes several studies of uniprocessor and multiprocessor bioinformatics applications.

Several bioinformatics benchmarks suites were proposed very shortly after the first release of BioBench: Li et al.[57] proposed a similar benchmark suite (BioInfoMark) that is somewhat broader in scope, and presented a detailed characterization suite which was done using a very similar methodology to ours. Bader et al. also presented a benchmark suite (BioSplash) in [14], which included some parallel applications and somewhat comprehensive workload characterization analysis. The BioSplash and BioInfoMark efforts were combined in late 2005, giving way to a more streamlined benchmark suite named BioPerf, described in some detail in [13].

3.6 Concluding Remarks

In this chapter of the dissertation, we briefly identified and described important computational biology application categories and presented BioBench, a benchmark suite of bioinformatics applications that represents relatively mature application classes with reference data that closely parallels real usage. BioBench applications and reference input data have been made available to researchers and was received with interest. In addition to allowing researchers to evaluate their systems using bioinformatics applications, BioBench also helped spur new benchmark suites for bioinformatics, and several such suites followed our work. We believe BioBench filled an imminent need for a well-defined set of benchmarks covering an important emerging class of applications.

Our evaluation of BioBench components validated our intuition that bioinformatics applications have characteristics that distinguish them from traditional scientific computing applications characterized by SPEC FP benchmarks. Bioinformatics applications evaluated in this study displayed almost no significant floating point instructions and higher ILP while having basic block lengths closer to SPEC FP benchmarks than SPEC INT, implying similar regularity in distribution of branches. These findings indicate that bioinformatics applications stand to benefit from future architectural features such as increased memory bandwidth and wider execution units to exploit their high ILP. In addition, some of these applications might benefit from prefetching as well.

The applications studied in this chapter are single-threaded, and some applications do not even have multithreaded counterparts. Given the current trends in microarchitecture, we expect this situation to change soon: Bioinformatics researchers will certainly

move quickly to take advantage of chip multiprocessor(CMP) architectures and parallelize important bioinformatics applications. We believe that the performance characterization work presented here will still be pertinent to evaluating single-thread performance of bioinformatics applications on future CMPs; and parts of the characterization data like instruction profiles might be helpful to guide future performance characterization studies of multithreaded bioinformatics applications. Overall, we believe that our findings from this characterization study paved the way for deeper analysis which formed the basis of the rest of this dissertation.

Looking ahead, we plan to expand BioBench with benchmarks from several other emerging bioinformatics application domains in its next revision. Considering the parallelism available in bioinformatics workloads, a parallel version of BioBench would be a very valuable tool for studying the characteristics of these codes on multiprocessor systems and clusters, and such a version of BioBench is among our plans for future work in this field. In addition, we will be conducting studies on different levels of parallelism available in bioinformatics applications by studying BioBench components in detail to evaluate how such applications can be accelerated using thread-level parallelism techniques.

Chapter 4

Memory System Performance of Bioinformatics Applications

4.1 Overview

The importance of bioinformatics applications has increased dramatically in the recent years, mostly owing to the availability of massive amounts of genetic data from the research community as well as the emergence of new techniques to use this data. Fast and efficient utilization of this wealth of new genetic information holds many promises, some of which are new therapies for many previously intractable diseases, accelerated drug discovery schedules and more resistant, high-yield crops. Considering the immense potential impact of the field on science and economy alike, we expect bioinformatics workloads to be among the most important scientific computing workloads in the foreseeable future.

Many bioinformatics applications, particularly those in the genomics domain; involve searching large volumes of genetic data for patterns. The amount of such data in public and private databases is increasing at an almost exponential rate; and this increase is expected to accelerate further in the near future. As of August 2005, the total amount of DNA and RNA data in the three leading public databases (GenBank, EMBL-Bank and DNA Data Bank of Japan) exceeds 100,000,000,000 bases, encompassing 165,000 different organisms[1]. The bad news for researchers eager to sift through this data is the fact that available memory bandwidth and latency in recent computer architectures is not exhibiting a trend similar to that of available biological data. Despite significant achieve-

ments in process technology, the gap between DRAM latencies and operating frequencies of modern microprocessors persists. Aptly named ”the memory wall” [100], the limitations imposed by this gap mean that memory access time is likely to remain as an obstacle on the path to higher performance. The adverse effects on application performance will be even more pronounced for workloads which operate on large amounts of data, such as bioinformatics applications.

To improve the performance of bioinformatics applications; a thorough understanding of memory access characteristics and cache performance of such workloads is therefore crucial. While general performance characteristics of some of these workloads have been studied in some detail on a per-application basis, very few results have been presented to provide insight into the memory performance characteristics of a wide range of bioinformatics applications. In this chapter of this thesis, we aim to present a quantitative study of memory footprints of bioinformatics applications sampled from a relatively comprehensive range of application domains. The remainder of this chapter is organized as follows: We begin by presenting a very brief overview and descriptions of the workloads we studied in Section 2. In Section 3 we outline some of the previous work related to the area, and Section 4 describes our experimental methodology. We present our results in Section 5, followed by our conclusions.

4.2 Applications

Over the last two decades, the scope of bioinformatics has widened to encompass many different application domains; and an exact and concise description of the field

remains elusive. Generally defined, bioinformatics is the application of information technologies to biology with the goal of understanding how the basic building blocks of living organisms develop and interact with each other. A more detailed introduction is provided in the first chapter of this thesis. For a more thorough description of bioinformatics concepts, common applications and algorithms, we refer the interested reader to comprehensive introductory works such as [59], [26] and [34].

In order to reflect the diversity of the field, the bioinformatics benchmarks used in this study were chosen from the BioBench benchmark suite [11] described earlier in Chapter 3, which contains applications from a reasonably diverse set of bioinformatics application domains. The input parameters and data sets of the original BioBench distribution were used without modification.

4.3 Related Work

While computational biology and bioinformatics have been active research topics for a long time; comprehensive performance evaluation studies of diverse bioinformatics applications have been absent from the literature until recently. It should be mentioned that earlier work describing individual algorithms and bioinformatics applications has usually included some coverage of cache performance characteristics.

More recently, there has been a series of proposals of bioinformatics benchmark suites and accompanying workload characterization efforts with varying levels of detail. Albayraktaroglu et al.[11] described the benchmark suite used in this study and presented a workload characterization of bioinformatics applications, including a basic cache per-

formance analysis and a comparison to both SPEC INT and SPEC FP suites. They articulate that bioinformatics applications exhibit cache miss rates between those of SPEC INT and SPEC FP benchmarks in general. A later paper by Li et al.[57] describes and characterizes a similar benchmark suite. According to their analysis, bioinformatics workloads have a lower average cache miss rate than that of SPEC integer benchmarks. It should be added, however, that their study excludes the much more memory intensive SPEC FP benchmarks from their comparisons. Bader et al. present a comprehensive benchmark suite in [13], without any memory performance analysis. In another study [14], Bader et al. present and characterize a bioinformatics benchmark suite which includes parallel applications. This study presents detailed memory system performance data covering L1 and L2 cache miss rates, TLB miss rates and number of memory accesses among others. Jaleel et al. studied the last-level cache access characteristics of several parallel bioinformatics applications in [45].

Memory access characteristics of more traditional applications have been studied in greater detail. Studies covering the SPEC benchmark suite[84], database workloads [16] [94], and multimedia applications [86] provide extensive insight into the memory access characteristics and performance of a wide spectrum of workloads.

4.4 Methodology

To obtain a comprehensive understanding of the memory behavior of bioinformatics workloads, we wanted to study the impact of different cache sizes on application performance. The two prevalent methodologies for these kind of studies are execution-based

and trace-based simulation. Execution-based simulation relies on a detailed functional model of the processor to capture the stream of memory accesses from the application under study. The performance problems inherent in this method led to the use of trace-based simulation; which involves generating and collecting memory address traces from an application using methods like hardware probes or instrumentation. These traces are then fed into a reconfigurable cache simulator. While collecting memory traces is practical for small computational kernels, benchmarks with small memory footprints and representative program slices; this method was not suitable for studying complete runs of the large benchmarks like those in this study. Therefore, the sheer size of the data sets used by the applications that we studied necessitated the development of a cache simulation framework that could cope with the large number of memory accesses.

We used the PIN binary instrumentation framework [58] to develop simCMPcache, a configurable cache simulator for the x86 architecture that does not require trace collection. In addition to being fully configurable to test different cache parameters such as size, associativity, and allocation/replacement policies; simCMPCache can also provide information on the total amount of data shared between different threads of an application. In the next section, we will describe simCMPcache briefly.

4.4.1 simCMPcache Cache Simulation Framework

PIN [58] is a dynamic binary instrumentation tool which provides the capability of running applications in an instrumented mode similar to a virtual machine. The current version of PIN supports a variety of Intel architectures including IA-32(32-bit x86), which

is the architecture we used in this study. PIN is similar in design and concept to the earlier ATOM toolkit for DEC/Compaq Alpha architecture. Similar to ATOM, PIN provides a complete infrastructure for writing program analysis tools called PIN tools.

A PIN tool is a C/C++ application that uses the PIN API to insert instrumentation calls in arbitrary points chosen by the programmer in a program instruction stream during execution. The PIN API provides full access to the internal data structures of the execution environment, providing a flexible mechanism for analysis tasks like counting certain types of instructions, or triggering different functions based on the type of x86 instruction encountered. Routines using the API for purposes like these are called instrumentation routines in PIN parlance. Instrumentation routines interact with analysis routines, which are called by the instrumentation routines in runtime to do the actual analysis. For example. a user can specify an analysis routine called *saveBranchAddress()* to save the branch address every time an unconditional branch instruction is encountered.With such comprehensive access to the instruction stream during actual execution on a real platform, PIN allows computer architects to design and implement a large variety of analysis tools to conduct microarchitectural studies. PIN also supports the instrumentation of multithreaded applications. PIN automatically detects the creation of new threads and allocates contexts for these threads. Analysis results from these threads can then be tagged with unique thread IDs assigned by PIN. We used a PIN-based cache simulator, **sim-CMPcache**, developed by Jaleel et al.[45] for their work on last-level cache performance analysis of bioinformatics benchmarks. This simulator uses two instrumentation routines named *Instruction()* and *Trace()*; which are used to identify new instructions and new basic blocks(traces) respectively. The *Instruction()* routine checks the new x86 instruction

to detect whether it is a memory reference or not. In the case that the instruction is indeed a memory access, then the thread ID, access type (read or write), the effective address and size are used as input to the resulting call to the main analysis function, *CacheLookup()*. This routine executes a cache lookup from the built-in configurable cache model.

Our cache model can implement a two or three-level cache hierarchy with as many caches per level as the number of cores in a CMP. In case of a CMP cache model, each level can be configured to be shared or private. If the workloads to be analyzed are multi-threaded, an invalidate-based MSI coherence protocol is used. In addition, simCMPcache can also model a TLB and provide statistics on TLB accesses.

simCMPcache is a very flexible and powerful cache simulation framework that harnesses the capabilities of the PIN framework to eliminate the large storage requirements and trace collection efforts associated with trace-driven cache simulators. Our use of simCMPcache allowed us to evaluate a large number of simulations with varying cache sizes to determine the memory footprints of bioinformatics workloads.

4.5 Experimental Results

Our consultations with practicing bioinformaticists suggested that bioinformatics applications in production environments were typically optimized to the highest extent possible; therefore we opted to optimize the benchmarks to a reasonably high level. All benchmarks have been compiled using gcc 3.3 at the optimization level -O3 and statically linked. The gcc compiler suite supports a host of optimizations for the x86 architecture like loop unrolling, and these have been enabled where applicable. Two of

the BioBench applications (BLASTN and PROTPARS) could not be executed in our binary instrumentation-based cache simulation framework, and had been excluded from this study. All of the remaining workloads were ran to completion using the the set of input data provided with the BioBench distribution. We used a "compute-farm" composed of systems based on various Intel Pentium 4 CPUs (running the same version of Linux) for our measurements. Using such a cluster allowed us to cut down the data collection time drastically; especially considering the large number of different cache size configurations that we used in our studies. The benchmarks were either single-threaded from the start or had multithreading disabled to make compilation on multiple platforms; and the simultaneous multithreading (SMT) support of the Pentium 4 architecture did not play a role in our measurements.

Many bioinformatics applications are known to be memory bound, and cache size can be a very important factor in the performance of such workloads. Our previous analysis of BioBench benchmarks showed that the instruction cache miss rates of many of these applications were in many cases negligible, and L1 data cache miss rates were less than 3% in average. To study the impact of L2 data cache size on performance, we used simCMPcache to vary the size of the L2 cache from 64KB to 32MB in increments of 64KB; and plotted the results. Our analysis methodology uses the **stack distance** method described in [61] to simulate multiple cache configurations in the same execution of the simulator: the 64KB configuration is direct-mapped and the next configuration (128KB) is 2-way set associative, and the associativity continues to increase as the cache size increases. We used a line size of 64 bytes, and modeled a non-inclusive, write-through, write-allocate cache with an LRU replacement policy. We chose to use misses per 1000

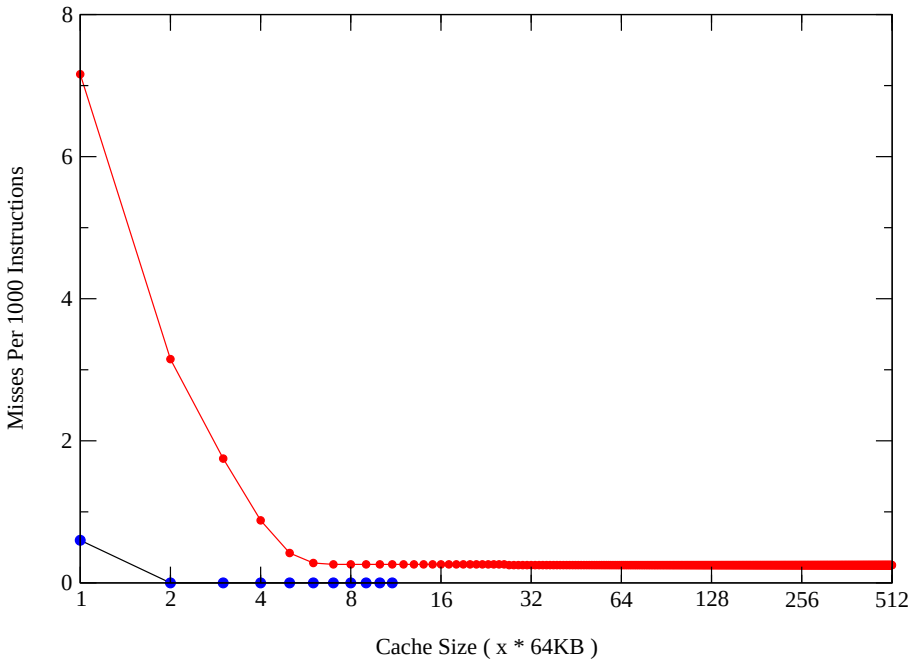

Figure 4.1: BLAST

instructions for our cache miss rate statistics as this metric is a more appropriate choice for the out-of-order Pentium 4 cores used in cluster we used to obtain our simulation data.

In general, the three sequence alignment applications we studied displayed similar memory footprint characteristics. The working set of BLASTN is around 512KB as seen in Figure 4.1. Figure 4.2 and 4.3 show the variation of L2 data cache miss rates of FASTA_DNA and FASTA_PROT respectively. The cache miss rate of FASTA_DNA is already at a fairly low level with minimal L2 cache size, and the working set seemed to completely fit in a 512KB cache. As the cache size increased to the vicinity of 2MB, the miss rate of this workload decreases to literally zero. The behavior of the protein sequence alignment counterpart is similar, and the working set appears to be around 256KB for that application.

Despite having a fairly large input data set, CLUSTALW has a very low cache miss

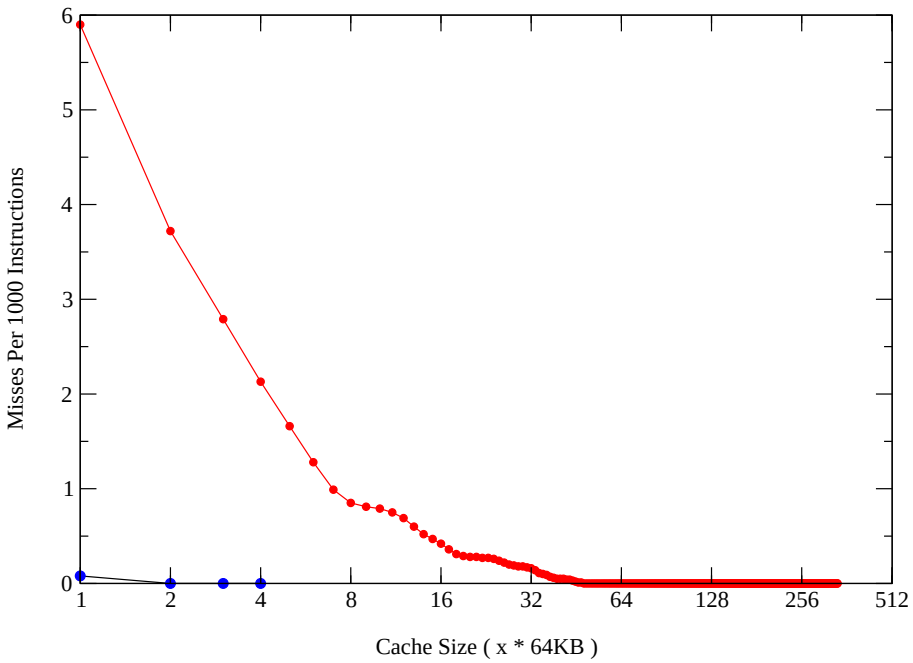

Figure 4.2: FASTA DNA

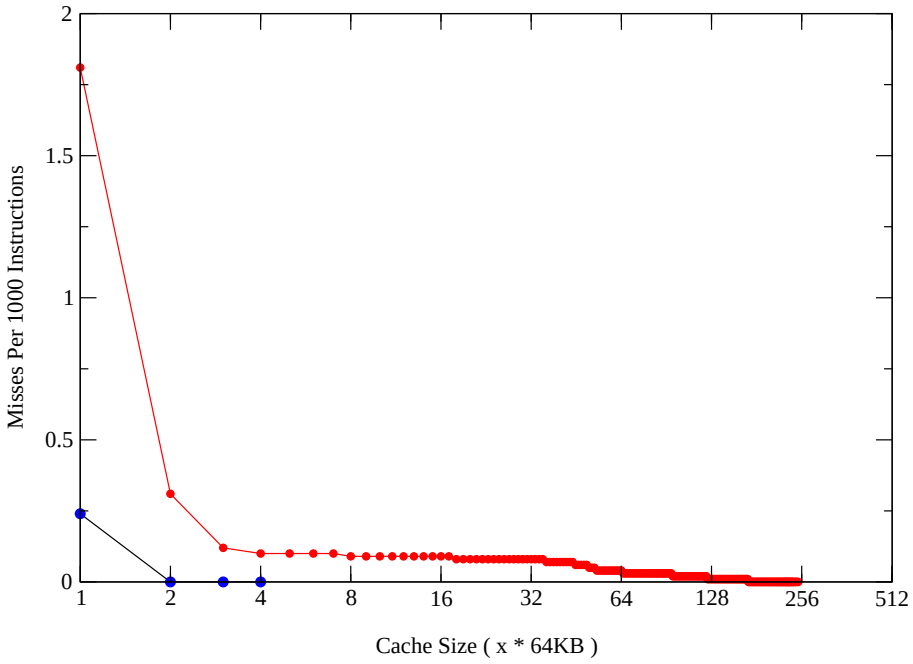

Figure 4.3: FASTA PROT

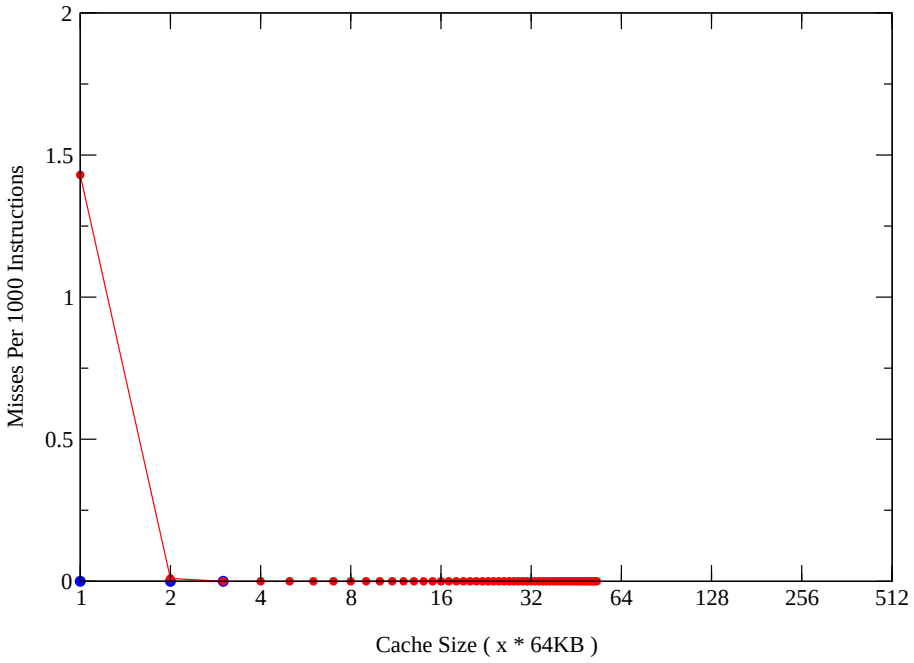

Figure 4.4: CLUSTALW

rate suggesting that this application has very high levels of data locality and a very small working set as seen in Figure 4.4. For the input data we used in BioBench, a 128KB cache was sufficient to fully contain the working set of CLUSTALW. CLUSTALW also had some of the lowest L1 and L2 data cache miss rates in our studies in Chapter 1 along with a relatively high IPC,

One of the few truly CPU-bound applications in the BioBench suite, HMMER appeared to have a primary working set that fit in approximately 256KB. The miss rate graph, seen in Figure 4.5 has a second plateau around 2MB, suggesting a secondary working set. The first working set is most likely the memory necessary to contain the in-memory representation of the HMM (Hidden Markov Model) data structure. The second inflection point corresponds to a point where the cache is large enough to contain both the HMM and the protein sequences to be searched against and the miss rate curve

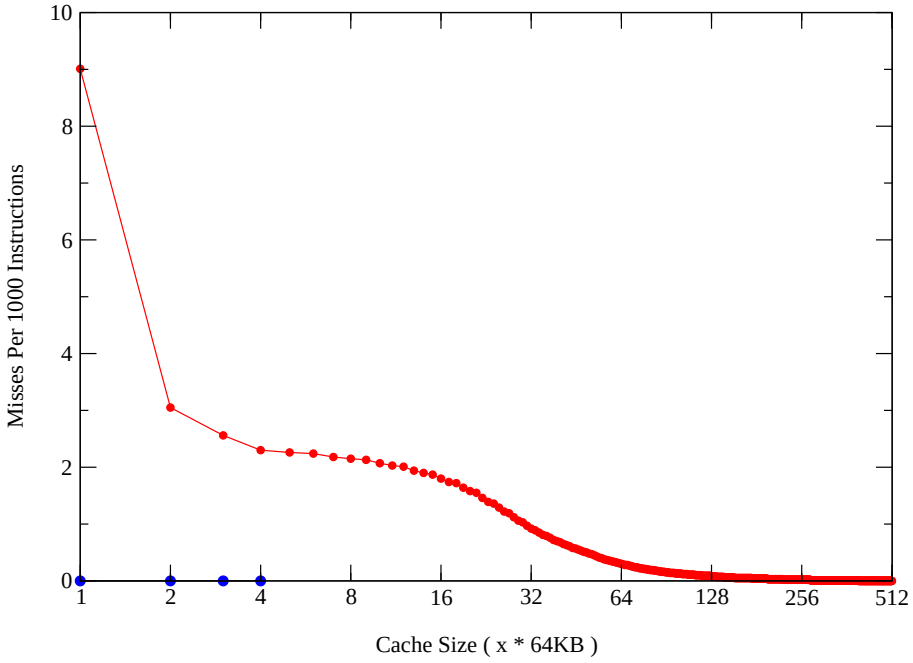

Figure 4.5: HMMER

tapers off.

In our previous analyses, TIGR was the BioBench application with the highest L1 data cache miss rate and the second highest L2 miss rate. Our findings in this study yielded similar results: We observed that the memory footprint of TIGR was around 4MB(Figure 4.6. Around 10MB, the cache miss rate of TIGR stabilizes to a value close to 10 misses per 1000 instructions; and increasing the cache size to 32MB does not provide a significant improvement.

MUMMER requires a cache size of around 64MB to decrease the miss rate to a relatively high figure of 7-8 misses per 1000 instructions (Figure 4.7; which means MUMMER will keep incurring higher miss rates than other bioinformatics applications even on the latest generation of chip multiprocessors such as the Intel Itanium 2 with its 24MB shared last-level cache.

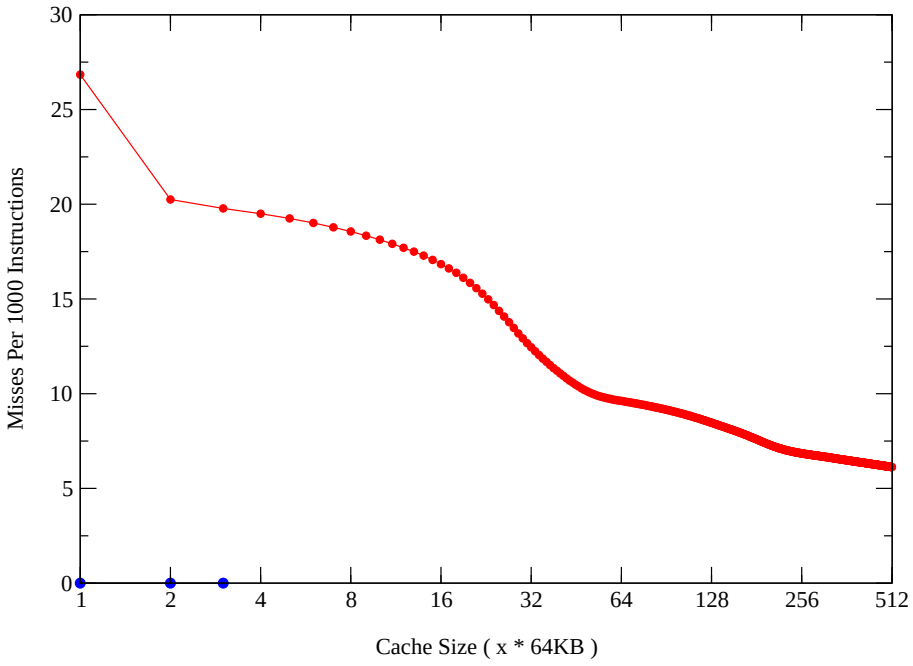

Figure 4.6: TIGR

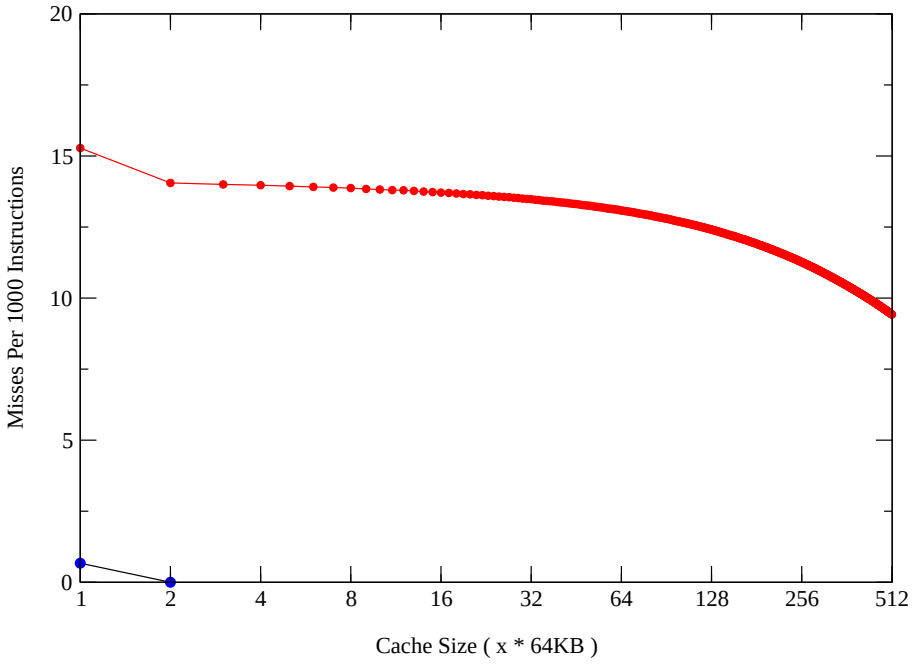

Figure 4.7: MUMMER

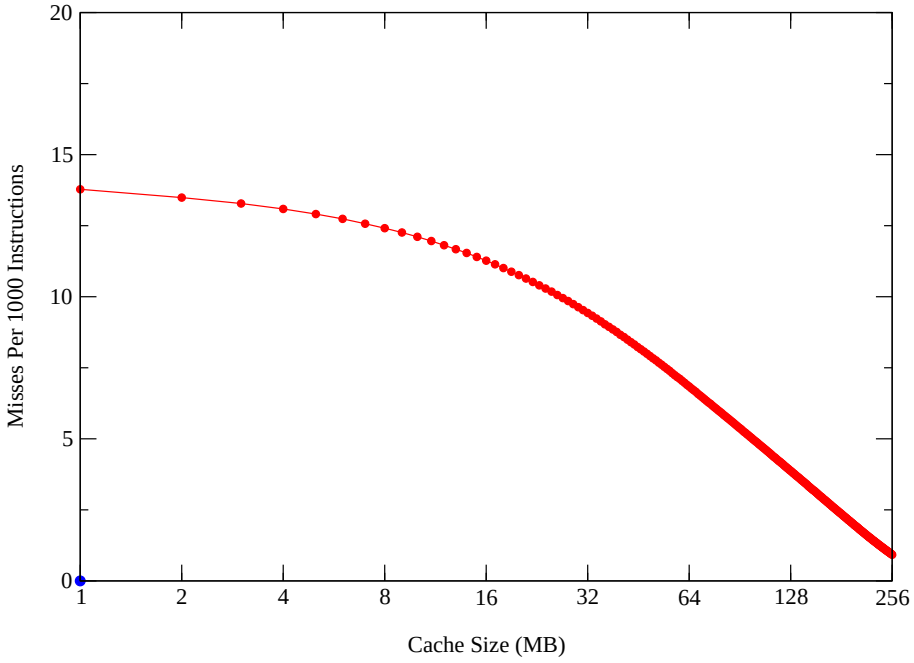

Figure 4.8: MUMMER256

Faced with the very high miss rate of MUMMER, we had to run a second experiment with larger cache configurations in order to understand its working set characteristics better.Figure 4.8 shows the results of this experiment, where the cache miss rates of MUMMER can be observed to decrease below 5 percent only when the cache size is increased to more than 64MB. The size of the working set seems to be close to 256MB. The large memory footprint of MUMMER can be attributed to the large tree structures that its main algorithm sets up in memory, and has been continually addressed by its developers in the form of new versions with lower memory utilization.

4.6 Conclusions

In this chapter, we used a binary instrumentation-based cache simulation framework to collect and analyze L2 data cache performance data for bioinformatics applications in

the BioBench suite. We observed similar cache access characteristics in sequence analysis applications such as FASTA and BLAST, which validated the findings of our previous work. Our simulation framework allowed us to analyze a large number of cache configurations and sizes, and our observations expanded our knowledge and understanding of BioBench applications. Most importantly, our more detailed analysis of the memory access characteristic of HMMER motivated us to concentrate our efforts on this important, computationally intensive benchmark. The "two-level" cache footprint of HMMER suggested that it would be possible to fit its working set in the confined memory space of the SPE local stores with frequent DMA transfers; while our earlier findings of a single dominant function within HMMER suggested it would provide a good candidate for SPE implementation.

Chapter 5

A Case Study: Protein Profile Searching on the Cell Broadband Engine

5.1 Introduction

In the previous chapters of this thesis, we characterized the performance of computational biology workloads in detail and elaborated on different aspects of these characteristics. As we have discussed in the introduction chapter, the amount of biological sequence data collected by researchers is already at staggering levels and keep increasing steadily. With the imminent introduction of even more sophisticated applications like system biology and protein folding/prediction applications, the computing performance demands of the emerging field of life sciences and bioinformatics will increase dramatically in the years to come.

While the demand for higher computer performance is increasing, the last few years have seen single core microprocessor performance reach a standstill due to a complex combination of many factors. Increasing power density and diminishing returns from deeper pipelines all but eliminated higher clock frequencies as a relatively effortless means of reaching higher performance levels. As a result, the focus of technical innovation has shifted to chip multiprocessing (CMP) architectures. While the first examples of these multicore processors have been fairly conservative designs similar to existing SMP architectures, processor architects have started to experiment with new, more unconventional multiprocessor architectures in an effort to bring much needed innovation to the

marketplace and enable higher performance levels.

The Cell Broadband Engine (Cell BE) architecture is one of the most novel entrants in this race. A joint project of Sony, Toshiba and IBM, Cell BE is a heterogeneous multiprocessor which combines a two-way simultaneous multithreading PowerPC processor core with eight DSP-like SIMD processing units (SPEs) and a high-bandwidth memory subsystem. Initially slated for use in the Sony Playstation 3 game console, Cell BE has features like a high-performance double precision floating point unit which suggest that the architecture has been conceived with future applications outside the games/entertainment domain in mind.

New microarchitectures like the Cell BE might be the key to enable even higher levels of performance for resource-intensive computational biology applications of today and tomorrow. We believe that the Cell BE architecture is particularly suitable for such workloads with high degrees of parallelism at different granularity levels. As a case study to evaluate the use of Cell BE in solving bioinformatics problems, we chose one of the most CPU-intensive workloads that we studied earlier for porting to the Cell BE. The *hmmsearch* protein profile searching application from the HMMER suite exhibits several characteristics that we believe makes it a good candidate for a Cell BE application:

- As seen in Figure 3.7, most of the execution time of HMMER is spent in a single function, providing the possibility of using the Cell BE SPEs as function offload engines. As we will elaborate further later in this chapter, the ”SPE as offload engine” concept is probably one of the most intuitive programming models for porting applications to Cell BE. In this model, applications are modified by separating com-

putationally intensive functions and optimizing them for the SIMD architecture of the Cell BE SPEs. The scalar processor in the Cell BE (the PPE) can execute the rest of the program in addition to managing data transfers between the main memory and the SPEs.

- Due to the nature of the algorithm (which operates by processing a relatively small data structure repeatedly with many sequences), there is a significant level of data parallelism in HMMER that can be exploited to keep the SPEs busy. If the DMA operations can efficiently be overlapped with computation, this could allow fairly high performance levels.

- Our previous analysis suggests that the HMMER application seems to be largely CPU-bound. Despite the impressive memory bandwidth of the Cell BE, memory-bound workloads are probably harder to port to the Cell BE due to the lack of large amounts of shared memory, and CPU-bound applications such as HMMER are better suited for the Cell BE SPE implementation.

In this chapter, we evaluate the performance of the Cell BE architecture running the HMMER protein profile search workload. We describe our modifications to HMMER to port it to the Cell BE architecture; and present our results. Our HMMER implementation for the Cell Broadband Engine, Cell-HMMER, provides up to a 27.98x performance advantage over a recent dual-core x86 microprocessor. Even when Cell-HMMER running on a single Cell BE processor is compared to a 2-way SMP system with two such dual-core processors, Cell-HMMER runs up to 14.13x faster.

The remainder of this chapter is organized as follows: Section 2 provides an in-

troduction to the background to the subject by describing the basic concepts of hidden Markov models, their use in bioinformatics, the HMMER suite, and the significance of the Viterbi algorithm. In Section 3, we present a brief outline of the Cell BE architecture, programmability issues and programming models. We describe our Cell BE implementation of HMMER and our experimental methodology in Section 4, followed by the experimental results in Section 5. Some of the related work on the subject are described in Section 6, and we close the chapter with our conclusions in Section 7.

5.2 Background

Some of the most important problems in modern bioinformatics involve determining the functionality of proteins, which are complex organic compounds that carry out many essential tasks in living organisms.While the complete structure of a protein is characterized by a combination of features such as its shape, the primary feature of the structure of a protein can be simplified as a string of letters describing different amino acids. Based on the assumption that structurally similar proteins have similar functions, one way of identifying and classifying proteins is comparing protein sequences with the contents of databases that contain protein sequences of known structure and functionality.

The difficulty of such database searches lies in the fuzzy nature of protein sequences. The process of evolution has a fairly high degree of randomness, and while proteins of similar functionality are indeed similar in structure; there usually are slight differences between them. These differences usually manifest themselves in the form of additions or deletions of amino acids in the protein sequence. As a result, this variability

rules out using common string search algorithms for protein sequence searches. To solve this problem, researchers have proposed using probabilistic methods commonly used in the field of machine learning such as hidden Markov models, which form the basis of the application we have studied.

5.2.1 Hidden Markov Models(HMMs)

A hidden Markov model(HMM) is a statistical description of a system which consists of a number of connected states, each of which can produce observable outputs. The name ”hidden” refers to the fact that the outside observer does not know exactly what state produced a certain output, but can infer a probability for any outcome. In many application of HMMs, an HMM is typically used to represent a real-world process whose statistical characteristics are either well understood or can be distilled from a large body of observation data, even though the exact mechanism of the process might not be understood in its entirety.

A very simple HMM can be in the form of a set of N interconnected, distinct states which constitute a Markov chain. At regular time intervals $(t = 1, t = 2, ..)$, the state of the system undergoes a change and transitions to another state, or possibly the same state. Assuming that the HMM is not a constrained-jump model, the state transition can be to any state in the chain. Alternatively, one can specify rules on which transitions are allowed; resulting in a constrained-jump model. An output symbol O_t is emitted by the current state at time t. The symbol can be any one out of M different options. Following the description and the notation in [81], the HMM in this example can be described by

several key parameters:

- The number of distinct states in the system (N). These states will be named $S_1, S_2, ..., S_N$. Furthermore, the state at time interval t will be q_t.

- The number of distinct symbols that can be emitted by each state (M).

- The state transition probability matrix $A = a_{ij}$ where a_{ij} is the probability of a transition from state i to state j, or:

$$A = a_{ij}, a_{ij} = P[q_{t+1} = S_j | q_t = S_i], 1 \leq i, j \leq N \tag{5.1}$$

- The output symbol probability matrix $B = b_i(k)$ where $b_i(k)$ is the probability of emitting the symbol k when the current state is S_i.

- An initial probability vector π where $\pi(i)$ is the probability of starting at state i.

$$\pi_i = P[q_1 = S_i], 1 \leq i \leq N \tag{5.2}$$

After a time period of L intervals, the outputs from this HMM will form the observation sequence $O = O_1, O_2, O_3.., O_L$. At this point, we can use the HMM to answer the following questions:

- Given a sequence of observations $(O_1, O_2...O_N)$; what is the overall probability of this particular sequence outcome?

- Given a sequence of observations $(O_1, O_2...O_N)$; what is the sequence of states that is the most likely to have produced these observations?

The first question is aimed at computing the probability of the observation sequence being created by the particular HMM in consideration. Rabiner [81] describes this as the *evaluation problem* and outlines the Forward algorithm which can efficiently solve the problem. The solution to the evaluation problem can be used as a measure of the fit between the observation sequence and the model.

To compute the probability of any path through the Forward algorithm, we start with creating an array of per-state probabilities $prob_t(i)$ where $prob_t(i)$ is the probability that the current state is S_i at time interval t. These probabilities are set to 0 at the beginning of the algorithm with the exception of the first(initial) state, which should be set to 1 as the observations have to start at this point. Afterwards, the observations in the sequence are considered one by one. Following the notation we introduced earlier, we call the current observation O_j. For each state, the total probability of this symbol to be observed at time t is the product of the possibility of the current state s producing this particular symbol ($b_s(O_j)$) and the probability of the system to reach the current state s at time t:

$$prob_t(s) = b_s(O_j).\sum_{i-1}^{N}(prob_{t-1}(i).a_{is}) \tag{5.3}$$

For every observation symbol O_j ($1 \leq j \leq M$), this computation should be done for every state $s = S_k$ ($1 \leq k \leq N$). Once this part of the algorithm is complete, the overall probability of the observation sequence can be obtained by adding the per-state probabilities at the end of the observation sequence.

$$prob_{sequence} = \sum_{i=1}^{N} prob_M(i) \tag{5.4}$$

The second question aims to estimate the exact sequence of states that could have produced the outputs observed. It is possible that the same observation sequence might

have been generated by different state sequences, but these state sequences will have different probabilities of generating the observation sequence. The most likely state sequence, then, is the best estimate one can provide. Finding a solution to this problem effectively reveals the ”hidden” part of the HMM as it allows us to guess the path taken through the model. The most computationally efficient way of doing this is through the use of the Viterbi algorithm[81, 95], which is very similar to the Forward algorithm just described. Instead of using a summation operator to compute the total probability of arriving at a particular state, the Viterbi algorithm uses a maximum operator to find the most likely path:

$$prob_t(s) = b_s(O_j).\arg\max_{i \in states}(prob_{t-1}(i).a_{is}) \tag{5.5}$$

As in the Forward algorithm, this computation takes place for each symbol in the observation sequences and every state. At the end of this process, the most likely final state is simply the one with the highest probability, which is:

$$P = \arg\max_{i \in states}(prob_M(i)) \tag{5.6}$$

Once the most likely final state is found, the most likely path can then be computed by tracing the most likely predecessors at every step through the use of a process called *traceback*.

Both Forward and Viterbi algorithms are basically matrix-vector multiplications. In the case of the Forward algorithm, the inner loop operations describe in Equation 5.3 require $N + 1$ multiplications and N additions. Repeated over N states, the total number of multiplications and additions for each symbol is $(N + 1).N$ and N^2, respectively. Factoring in the M symbols in the observation sequence, the time complexity of the Forward

algorithm (and the very similar Viterbi algorithm) is $O(N^2.M)$. The storage requirements of the Forward and Viterbi algorithms are dominated by the probability matrix formed by the per-state probability array ($prob_t(i)$), meaning that the space complexity of these algorithms is $O(N.M)$.

5.2.2 HMMs in Bioinformatics

Starting in the last decade, the power and flexibility of HMM-based statistical machine learning techniques has been recognized by the bioinformatics community who adapted HMM techniques to various computational biology applications such as gene prediction, protein fold recognition and sequence alignment(These and other uses of HMMs in bioinformatics have been outlined in [20, 31]). The HMMER workload that we used in our study is one of the most common HMM-based applications in bioinformatics, and it uses ”profile HMMs” for analyzing protein families and classifying proteins by comparing them to statistical models of protein families.

The name ”profile HMM” refers to a specific type of HMM used to represent the common characteristics of a related family of proteins. Early work on profile HMMs and HMM-like models used relatively simple HMM representations to model protein motifs (common sequences preserved in similar proteins). Krogh et al.[52] were the first to propose a profile HMM in the currently used sense of the term, which utilizes delete and insert states that allow the addition and omission of amino acid symbols anywhere in the sequence. Eddy [31] differentiates between the later HMM representations and earlier ones by calling these ”profile models” and ”motif models” respectively. The Plan7

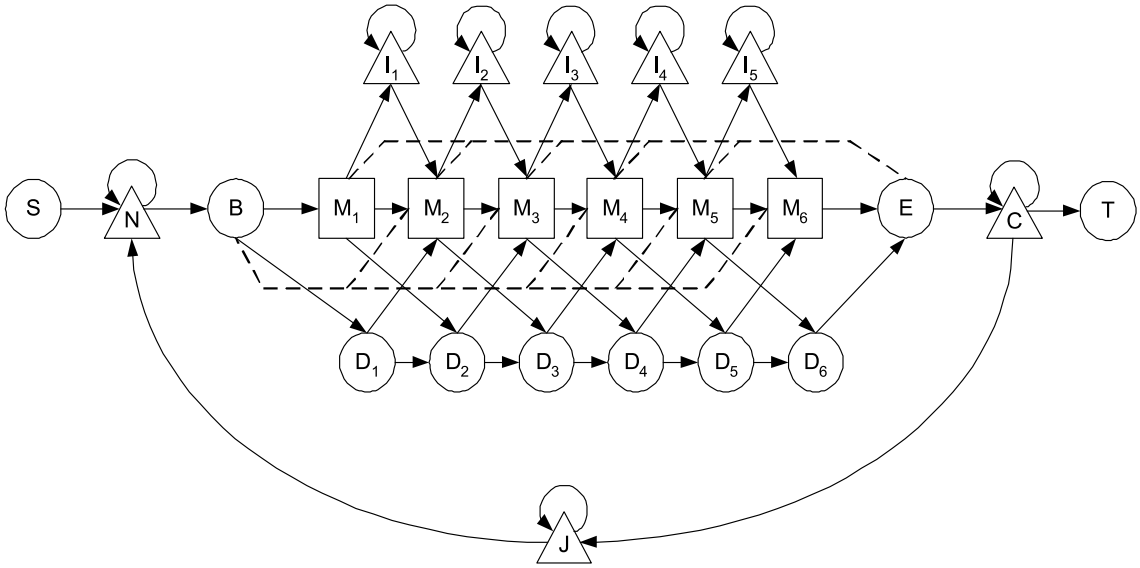

Figure 5.1: Plan7 HMM Example (motif length=6)

HMM architecture used in the current version of HMMER represents the final stage in the evolution of profile HMMs. Since the HMM architecture used in HMMER is central to our analysis of its performance in this study, we will briefly describe the Plan7 HMM architecture in the next section.

5.2.3 Plan7 HMM Architecture

The particular HMM template used in HMMER is called the Plan7 profile HMM architecture; and has unique characteristics which reflect the nature of the protein profiles it is used to represent. More specifically, the unique properties of the Plan7 model stem from the necessity of being able to handle gaps in alignment where the evolutionary process might have caused certain amino acid sequences to be deleted or inserted.

Figure 5.1 shows a 6-node (motif length=6) Plan7 HMM as used in HMMER. The basic states in the Plan7 architecture are match(M), insert(I) and delete(D) states. Each

match state is associated with an insert and delete state (with the exception of the very last match state); and each $M/D/I$ state triplet is called a "node". Each match state M can emit one of the 20-24 different amino acid symbols. Insert states also have the same number of possible outputs; while delete states do not emit symbols. Match states correspond to a one-to-one match between a particular amino acid in that location to the same amino acid in the HMM. Occasionally the HMM can shift to insert states and introduce sequences that do not match the model. In order to skip any number of match states, the HMM can shift to the chain formed by the delete states and follow this path until the match states where the sequence and the HMM converge again.

Every Plan7 HMM begins and ends with non-emitting begin(B) and end(E) states. The B state makes it possible to enter the main model at any match state through the use of $B \rightarrow M_i$ transitions, and similarly $M_i \rightarrow E$ transitions make it possible to exit the model from any match state. The rest of the states in the Plan7 HMM are special states (S, N, C, T, J) that are used for controlling algorithm-dependent properties of the model. S and T are the beginning and end points of the HMM. The N and C states can emit non-motif sequences before and after the motif, and the J state can emit non-motif sequences between two copies of the motif. The N,C and J states were added to the Plan7 HMM architecture to address the need for dealing with local alignments and multiple-hit alignments. [30] details the use of these states.

In the context of protein profile matching, the primary problem one would like to solve using an HMM is finding the likelihood that a certain protein sequence might be related to the protein family modeled by the HMM. As described before in the context of generic HMMs, this problem can be solved using the Viterbi algorithm. In this case, the

sequence of observations is the protein sequence (string of amino acids), the ”time” axis is the position of an amino acid symbol in the sequence. The HMM parameters are set by training the model using a suitable method as described before.

To illustrate the use of a Plan7 HMM for protein profile matching, we assume an HMM for a protein family P; and a protein sequence S of length N. One can then calculate the most probable path through the HMM that generates the sequence S by using the Viterbi algorithm. If we call the overall probability of this path $P(s)$; then one can compare $P(s)$ with the probability of S having been generated by a null model. If $P(s)$ is sufficiently higher than this probability; it can be concluded that the sequence S closely matches the HMM and might be related to the protein family represented by the HMM. Due to the existence of several different types of states, running the Viterbi algorithm on a Plan7 HMM is only slightly more complicated than running it on a simpler jump-constrained model we used earlier to introduce the algorithm. The model constraints greatly limit the number of incoming transitions to states: with the exception of the first M state, each M state has 4 incoming links, each non-emitting D state has 2, and each I state has 2 including itself. As a result, the inner loop of the Viterbi algorithm can be completely unrolled in implementations using the Plan7 HMM. A pseudocode implementation of the Viterbi algorithm (which omits the special states N,C,B,E and J for simplification) for a Plan7 HMM is given in Figure 5.2

The constraints on the number of incoming transitions to a state imply that the time complexity of the Plan7 Viterbi algorithm are somewhat less than that of the Viterbi algorithm on a non-jump constrained HMM. For a Plan7 HMM with N states and a sequence of length M, only a fixed number of multiplications and additions will be needed for a

```
/* Viterbi algorithm for Plan7 HMM */
/* N=number of states, L=length of sequence */

for  (i=1;i<=N;i++){
   for  (j=1;j<=L;j++) {
      /* Process M states */
   prob_m[i][j]=MAX({4 incoming transitions to match state})*bm[i][j];
      /* Process D states */
   prob_d[i][j]=MAX({2 incoming transitions to delete state});
      /* Process I states */
   prob_i[i][j]=MAX({2 incoming transitions to insert state})*bi[i][j];
 }
 process_special_states();
}
traceback(prob_m, prob_d, prob_i);
```

Figure 5.2: Pseudocode of the Viterbi algorithm main loop

full Viterbi computation, resulting in a time complexity of $O(N.M)$ instead of $O(N^2.M)$ for the "generic" Viterbi. Therefore the computational cost of the Plan7 Viterbi algorithm increases linearly with HMM or sequence length.

5.2.4 HMMER

HMMER[31] is a collection of biological sequence analysis applications using profile hidden Markov models. While the HMMER suite includes different applications for purposes like multiple alignment (*hmmalign*), protein domain searching (*hmmpfam*), and HMM database management (*hmmbuild, hmmcalibrate*); we chose *hmmsearch*, the most widely used and the most computationally intensive HMMER application, for our studies. *Hmmsearch* is used to search a protein sequence database against a single HMM representing the common characteristics of a protein family, generally with the purpose of finding sequences which may be related to the family modeled by the HMM.(Another application in the HMMER suite, *hmmpfam*, uses the same underlying algorithms but

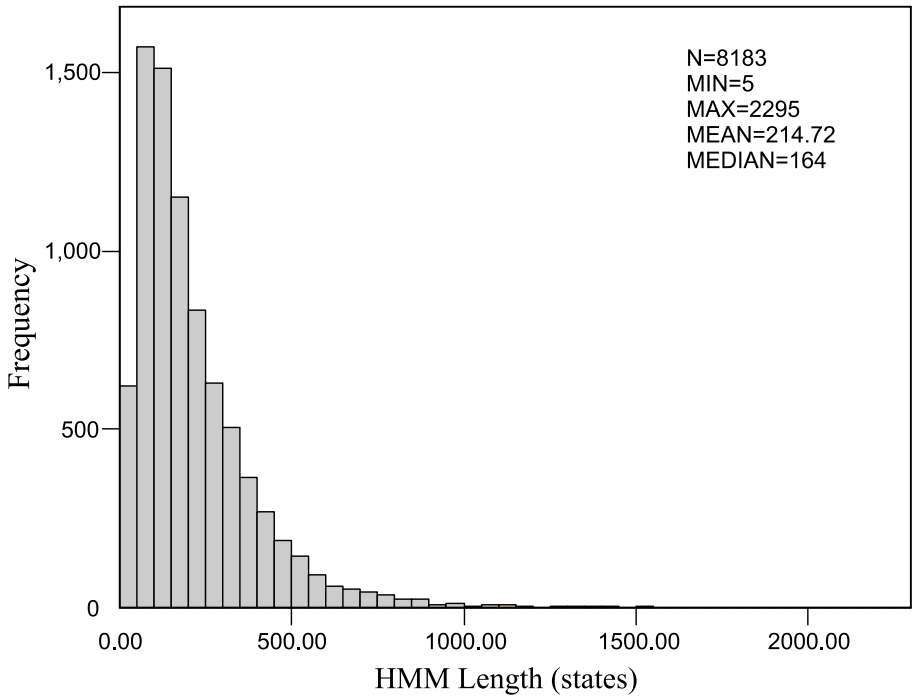

Figure 5.3: Histogram of HMM Lengths in PFAM Database

searches an HMM database against a single protein sequence.) Throughout this dissertation, the terms HMMER or ”HMMER workload” are used to refer to the *hmmsearch* application.

5.2.5 Input Data Characteristics and HMMER Performance

The limited capacity of the Cell BE SPE local stores necessitates careful planning and allocation of storage for the HMM, which will in turn have implications about program design and parallelization methodology. Our early work on porting HMMER to the Cell BE architecture suggested that the maximum length of the HMM we could fit on the SPE local store would need to be limited in the first version. In order to have a clear understanding of the nature of the input data and its impact on HMMER performance; we

first studied the characteristics of the PFAM protein profile database.

The HMM models used in our studies come from the curated Pfam[17] protein profile database, which contained HMM profiles of more than 8,000 protein families as of May 2006 and had a coverage of 75% of all proteins represented in the comprehensive SwissPROT protein database. A histogram of the distribution of HMM lengths in this database can be seen in Figure 5.3. The arithmetic mean of the lengths(number of states) for all HMMs in Pfam is 214.72, and the median is 164. The largest HMM in the database has 2295 states. The 99.5 percentile point corresponds to 982, meaning that 99.5% of all HMMs in Pfam have less than 982 states. Our early work on porting HMMER to the Cell BE architecture suggested that the maximum length of the HMM we could fit on the SPE local store would need to be limited in the first version. Considering that most typical HMMER searches use profiles from the Pfam database; we used these characteristics to guide our design decisions during the implementation of the Cell-HMMER.

HMMER search begins by loading the HMM data from an HMM database file into an HMM data structure, which contains transition probabilities for every state and emission probabilities for every emitting state as well as basic metadata about the HMM such as length (number of states, N) and name of the protein family being described. In order to simplify the Viterbi algorithm and improve its performance, HMMER converts and stores the HMM data in log-odds format, where all transition probabilities are converted into negative logarithms in order to change the multiplication operations into simple additions. After the log-odds conversion, HMMER allocates a re-sizable data structure large enough to hold the complete set of intermediate probabilities. This information is needed for the traceback step, and requires memory space on the order of $O(N.M)$ for each se-

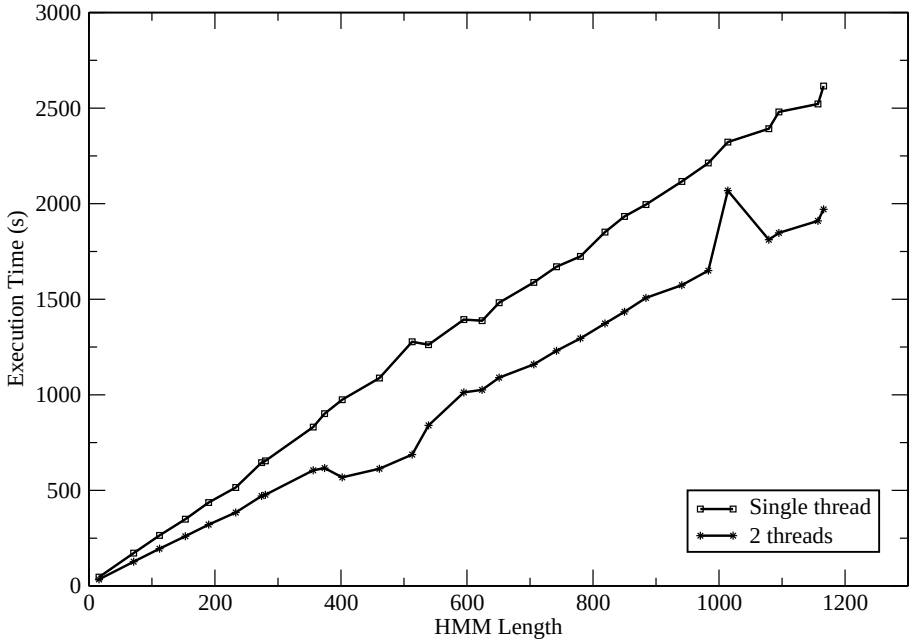

Figure 5.4: HMMER execution time vs. HMM length

quence of length M. HMMER then starts reading sequences from the sequence database, and executes the Viterbi algorithm for each sequence against the HMM. The process is relatively easily parallelizable as the sequences are completely independent units of work, and no data exchange between threads is needed throughout the computation(The standard HMMER distribution includes versions parallelized using both *pthreads* and MPI.)

Our previous analysis of the HMMER protein profile searching application as part of the BioBench suite showed that a significant portion of the execution time is spent in a single function which implements the Viterbi algorithm(Figure 3.7 shows that the *P7Viterbi()* function takes up 98.21% of the total execution time.). In order to quantify the impact of the variation of input HMM size on the percentage of total execution time spent in the Viterbi algorithm, we used the same set of 30 HMMs as in our previous experiment, and profiled(using *gprof*) the execution of HMMER during HMM searches

of the SwissPROT database against each HMM. The SMT feature of the Intel Pentium 4 CPU was disabled for this experiment. The resulting plot is shown in Figure 5.5. The Viterbi algorithm takes up more than 95% of the execution time for all HMM lengths up to around 800, and its share is higher than 90% for all HMMs used in the test. The share of the Viterbi algorithm is definitely more than 98 percent for HMMs between 50 to 500, where most HMMs(more than 93%) in Pfam lie according to our previous analysis of the database. It then seems to fall below 94% for HMMs larger than 900 states, and continues falling. Upon closer examination of the *gprof* output; we saw that the decrease in the execution time share of the *P7Viterbi()* function for larger HMMs was due to the increasing share of the *P7ParsingViterbi()* function, which partitions larger HMMs to run the Viterbi algorithm with limited system memory. Even for the largest HMMs, these two Viterbi algorithm-related functions completely dominate the execution time of the HMMER workload.

To analyze the impact of HMM length on HMMER execution time, we selected 30 different HMMs from the Pfam database and used HMMER to search the SwissPROT protein sequence database against each HMM on an 2.8GHz Intel Pentium 4 system running an unmodified version of HMMER 2.3.2. The lengths of these HMMs vary from 16 to 1166, covering most of the length range in the Pfam database. We measured the execution times of these searches for both single and dual-threaded operation modes of the Pentium 4 CPU using its 2-way SMT(simultaneous multithreading). The results, which can be seen in Figure 5.4, illustrate that the execution time increases almost linearly as the HMM size (number of states in the Viterbi algorithm) increases. Assuming that the execution time of HMMER is dominated by the Viterbi algorithm(an assumption we will show

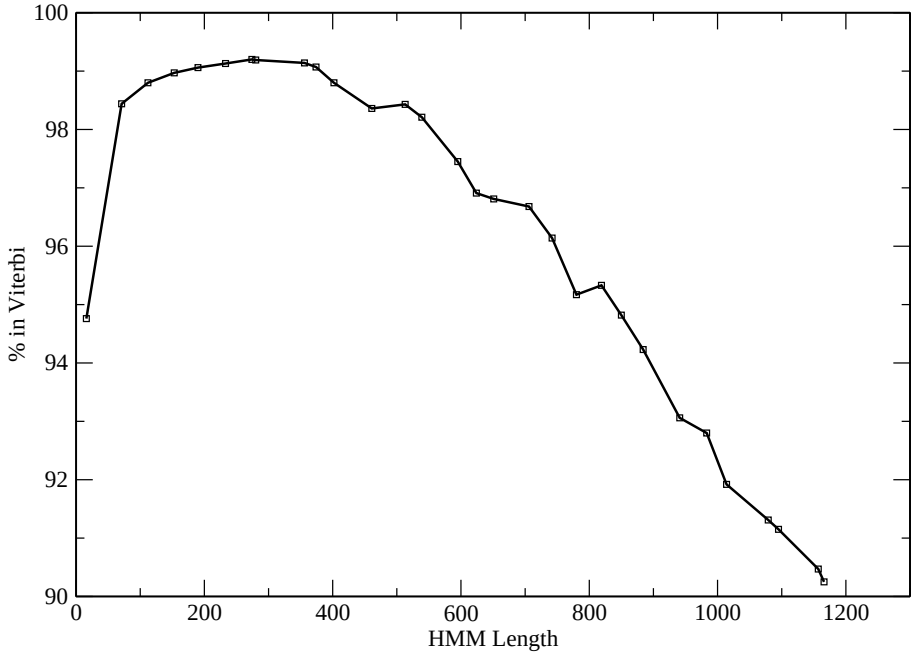

Figure 5.5: Percentage of time spent in Viterbi algorithm vs. HMM length

to be correct in our next experiment), this result verifies the linear relationship between HMM length and the performance of the Viterbi algorithm for Plan7 HMMs described earlier. The SMT feature appears to result in a 20-25% performance improvement for all HMM lengths in dual-threaded tests.

Finally, we implemented a random protein sequence generator to create random protein databases and search them against the same HMM to obtain the results shown in Figure 5.6, which shows the linear relationship between HMMER performance and sequence length.

5.3 Cell BE: A Novel Chip Multiprocessor Architecture

Faced with the challenges of increasing power consumption and diminishing returns on clock frequency increases, processor architects have recently turned to chip multipro-

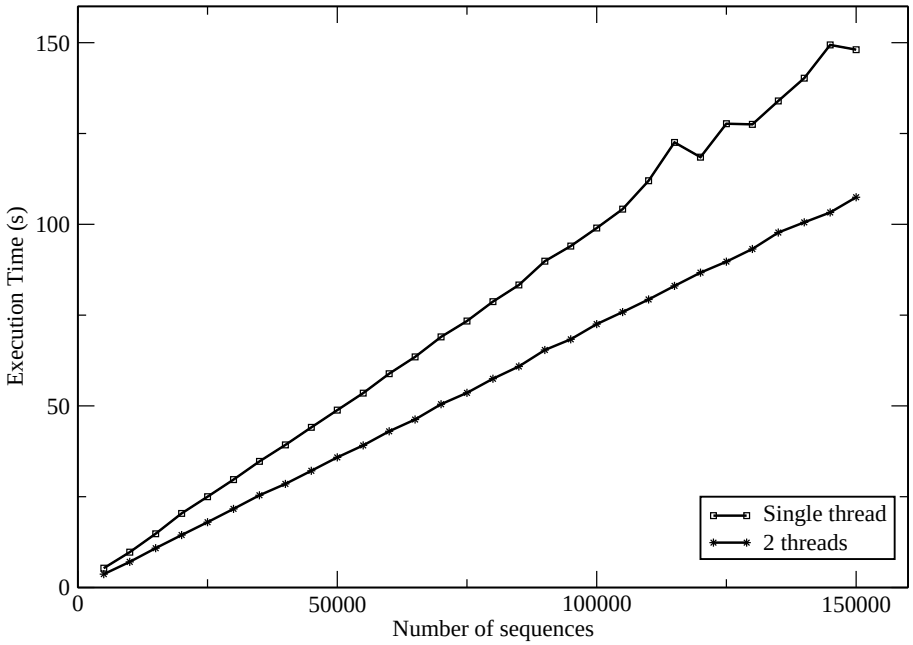

Figure 5.6: HMMER execution time vs. sequence database size

cessors (CMPs) to improve performance[88]. The advent of mainstream chip multiprocessors, which has long been anticipated by researchers[73], took place gradually; as companies first designed CMPs based on proven processor cores with little or no modification for CMP use. A later and more recent approach to CMP design is the use of relatively simpler, mostly in-order processor cores that maximize instruction throughput[28]

Designers of future CMP architectures will more than likely explore a larger portion of the CMP design space. We anticipate that the industry will continue increasing the number of cores in their future CMPs while adopting as yet unfinalized, novel techniques to improve single-thread performance using multiple cores. The importance of obtaining high single-thread performance in a CMT system is evident from the many different schemes that have been proposed for this purpose[24, 66]. Another possible future direction for CMP design could be the use of single-ISA heterogeneous mul-

tiprocessors where large out-of-order cores with high degrees of speculation are combined with several smaller, simpler in-order cores to maximize both single and multiple thread performance[54, 53]. Other possibilities include CMPs with coprocessors, which could combine high-performance CMPs with application-specific custom comprocessors through the use of high-performance I/O link technologies like HyperTransport (HT). Regardless of which CMP organization(s) will dominate in the longer term, we will likely witness many different CMP architectures in the next decade, some with novel features.

The Cell Broadband Engine (”Cell BE”) is one such CMP architecture developed with the collaboration of Sony, Toshiba and IBM. Developed primarily for the Sony Playstation 3 game console, the Cell BE will also be used in media and entertainment devices such as high definition TV sets. However, some features of the Cell BE architecture such as its double-precision floating point capability suggest that its designers had envisioned other roles for the Cell BE from the beginning, as these features are of little use in a game system. Before we explore the potential use of Cell BE for running a computationally intensive bioinformatics application, we will describe the architecture of the Cell BE in detail.

The Cell BE represents a radical departure from mainstream CMP design practices with its unique architecture. It is organized as a heterogeneous, multi-ISA multiprocessor with two different types of processor cores using different ISAs. One of these cores is the PowerPC Processing Element (PPE), which is a 2-issue, in-order, 2-way simultaneous multithreading PowerPC core that can act as a main processor or a controller for the rest of the CMP. The PPE implements the full 64-bit PowerPC instruction set alongside the full set of VMX (AltiVec) SIMD instructions; and is fully compatible with all PowerPC

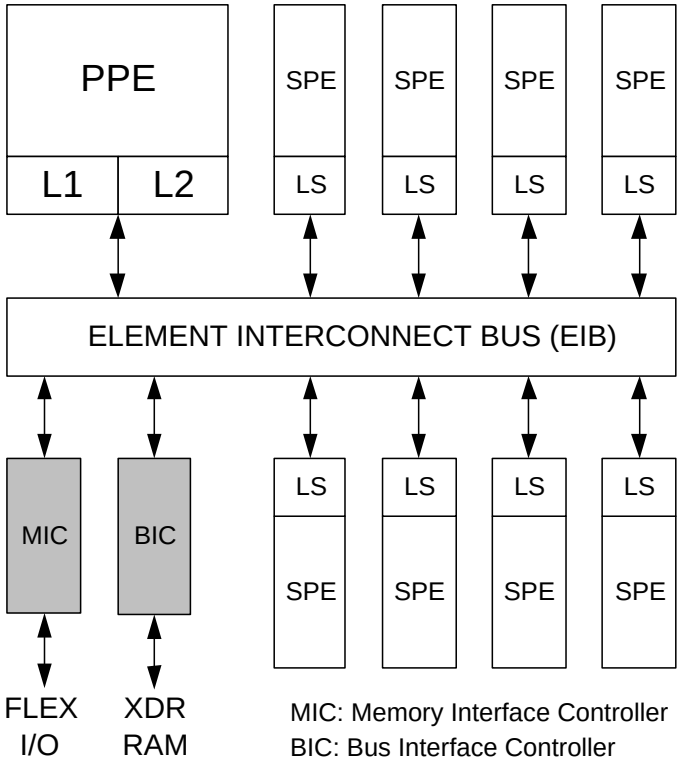

Figure 5.7: Cell Broadband Engine block diagram

applications. This feature provides the Cell BE with full backward compatibility with existing PowerPC software, an important feature to facilitate the adoption of the processor. The PPE has access to 32KB split L1 caches and a 512KB unified L2 cache.

The Cell PPE comprises of three separate units: The instruction unit (IU), fixed-point execution unit (XU) and the vector scalar unit (VSU). The functions of the IU involve instruction fetch, decode, issue and completion. Branches are also handled by the IU with the help of a branch predictor using a 2-bit by 4KB branch history table and 6 bits of branch history per thread. Four instructions per thread are fetched every cycle, and the IU can issue two instructions per cycle after dependency checking. Structural hazards do not allow the issue of two instructions to the same functional unit and some

other combinations of vector operations, but every other combination of instructions can be issued.

The XU is tasked with executing loads, stores and fixed-point operations. It consists of a high performance fixed point arithmetic logic unit, a load/store unit and a 64-bit, 32-entry register file. The XU load/store unit has an 8-entry miss queue and a 16-entry store queue.

The VSU handles floating point and vector (SIMD) operations. SIMD instruction set extensions have proven themselves as an effective way of increasing multimedia application performance and have been part of almost every major ISA for the last decade. The Cell BE architecture implements the full VMX SIMD ISA which allows operations on 128-bit words which can contain data elements of varying width (i.e. 1x128-bit, 2x64-bit, 4x32-bit, 8x16-bit or 16x8-bit elements in a single register/memory location). The VSU has a dedicated vector register file containing 64 x 128-bit registers. For floating point operations, the VSU has a ten-stage double precision (DP) execution pipeline and a FP register file containing 32 x 64-bit registers. DP instructions are handled by the floating point unit of the VSU whereas single precision (SP) floating point operations are executed by the vector units.

In addition to the PPE, the Cell BE architecture contains eight smaller processor cores. These processors are called Synergistic Processing Elements (SPE) and implement a completely new 128-bit ISA with a rich set of SIMD instructions. The SPEs can not access the main memory directly; but they have very low latency access to very fast, dedicated 256KB SRAM-based local memories called “local stores”. Data transfer to and from the main memory are accomplished through the use of fast DMA controllers

on the chip, the use of which can be initiated by either the PPE or the SPEs. The DMA mechanism maintains a coherent view of the memory and use the same page tables as the PPE; therefore addresses can be passed between PPEs and the SPEs. A typical SPE DMA data transfer moves 128 bytes at a time and SPE local stores have a 128-byte wide access port (in addition to a narrow port which handles 128-bit non-DMA accesses) for DMA access. The SPEs can work with any of the 128-bit vector element organizations mentioned before. Access to the local 128-bit registers generally takes 2 cycles, and access to the local store takes only 6 cycles.

Local stores are used as both instruction and data memories, which place important limitations on code and data size for the SPEs and impact programmability. For streaming applications similar to those in gaming and multimedia workloads that the Cell BE targets, the application kernels and data usually fit in the SPE local stores and the distributed nature of the SPEs make it possible to support a very large number of simultaneous memory transactions.

The Cell BE architecture had been designed to support a 2-way symmetric multiprocessing in a “glueless”(without requiring additional hardware) configuration. A Cell BE system configured in this way can make all 16 SPEs available to applications running on any one of the PPEs. A 4-way Cell BE configuration is also possible with the use of additional hardware, and these configurations are illustrated in Figure 5.8. Even larger Cell BE configurations are likely to be implemented in the future. IBM recently announced[51] that it will be building a next-generation supercomputer (named “Roadrunner”) with 16,000 AMD Opteron processors and an equal number of Cell BE processors for the Los Alamos National Laboratory.

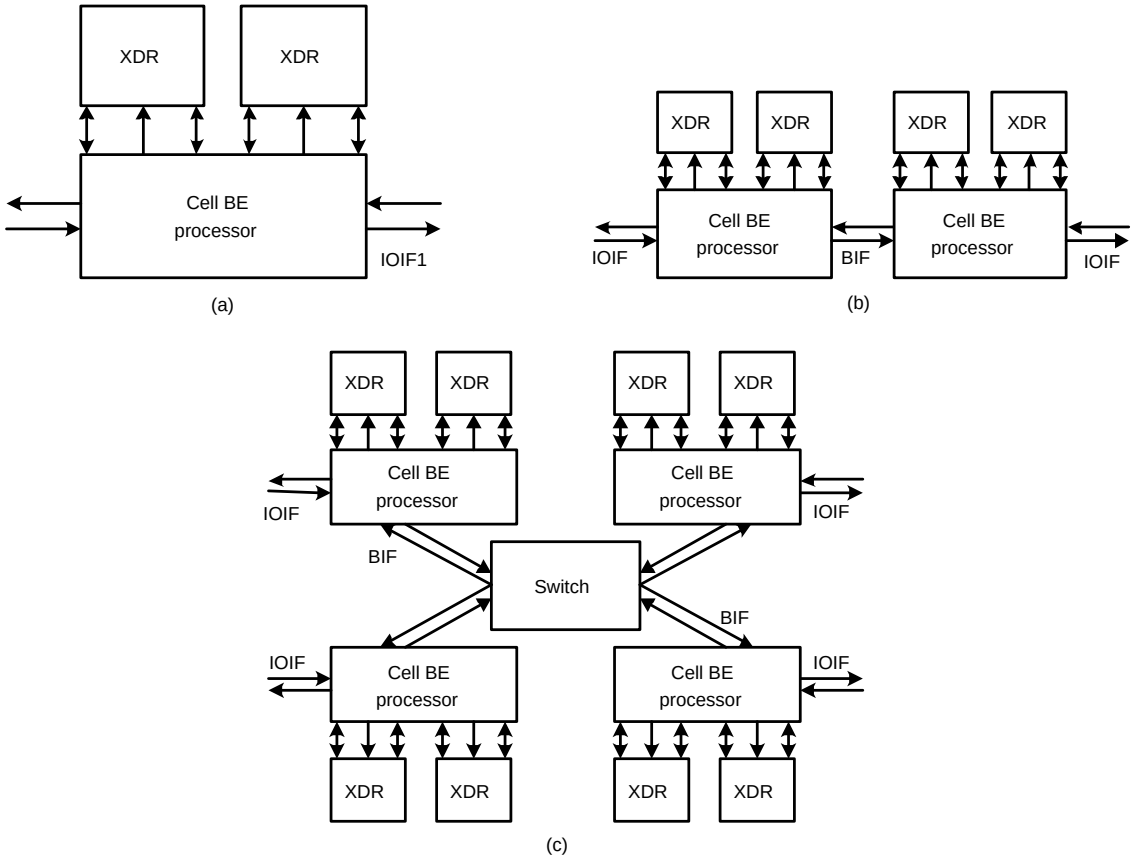

Figure 5.8: Cell BE configuration

A detailed look at the Cell BE architecture reveals its designers' desire to attack the multiple important challenges faced by computer architects in recent years. Arguably the most important of these is the increasing power densities of the processor cores. While advances in process technology made it possible to increase microprocessor performance by simply increasing clock frequencies over the last decade or so, the limitations of cooling technology made it impossible to continue this practice. Similar problems had been faced before during the era of bipolar transistors, and designers were able to overcome these power limitations by migrating to CMOS technology. This time there is no viable replacement for the CMOS process technology, eliminating the possibility of increasing

clock frequencies as a means of attaining relatively effortless performance gains.

With features such as the use of a relatively simple, in-order core for the PPE and extensive clock/power gating, the Cell BE multiprocessor design was conceived with low power consumption in mind. The PPE instruction pipeline is only 11 stages deep, enabling a power-efficient design that can be clocked at high relatively high frequencies without incurring excessive costs in power consumption. Kahle et al.[48] state that the original design goal was to have a delay of 10 FO4 throughout the processor, which was later adjusted to 11 because of area limitations.

The second significant obstacle on the path to higher microprocessor performance is the gap between DRAM access latencies and processor core speeds. Throughout most of the history of modern computer architecture, DRAM access latencies have lagged behind processor core speeds, mainly due to the differences between the process technologies required to build high-density DRAM circuitry and high-speed microprocessors. Since most of the DRAM access latency is RC delay which does not improve with scaling, and pipeline depths of modern microprocessor cores have been increasing; the DRAM access latencies are effectively increasing. As a result, DRAM latencies in recent microarchitectures have reached spectacularly high levels, such as 224 cycles for the Intel Pentium 4[5]. Barring an unprecedented breakthrough in memory technology, memory access latency seems destined to remain as a barrier to higher processor performance in the years to come.

The Cell BE design attacks the “memory wall” with multiple design features:

- A very high-bandwidth on-chip coherent bus (EIB): The Element Interconnect Bus

(EIB) provides a very high-bandwidth coherent memory fabric that allows high speed communication between the different processing units on the chip. The EIB can deliver a maximum bandwidth of 96 bytes per cycle, and allows the main memory to be accessed as a single address space by either the PPEs or the SPEs (for DMA transfers).

- High-performance XDR memory technology: The first examples of the Cell BE will use Rambus XDR DRAM modules. Two channels are supported, each of which can provide a maximum bandwidth of 25.6GB/s.

- Very fast SRAM-based local stores for every SPE: The use of dedicated 256KB SRAM-based local stores (LS) for each SPE reduces main memory traffic significantly and increases the maximum number of memory transactions in flight by offloading

- 128-entry register files in every SPE: The use of a very large (128 entries) and very wide (128-bit) register file provides the Cell BE compiler with greatly improved flexibility and reduces spills to the main memory. Unlike some other architectures which implement separate register files for general purpose and SIMD instructions, there is only one kind of register in the Cell BE SPE for both instruction types: all registers can be used to store integer and floating point values, and 128-bit SIMD vector elements of different element widths.

The idea of on-chip coprocessors to improve the performance of vector operations has been implemented on Sony's Emotion Engine processor, the CPU used in Sony Playstation 2 [70]. Emotion Engine utilized two vector processing units (VPUs) with

single precision FP units in addition to SIMD execution units; albeit with much smaller local stores (VPU0 had separate 4KB instruction and data memories, while VPU1 had 16KB for each). Some Cell BE design ideas like DSP-like processing units with fast local stores and the use of fast DMA controllers to shuttle data between the main memory and local stores might have originated in the Emotion Engine, considering that Emotion Engine was designed by the collaboration of Sony and Toshiba, two members of the Cell BE consortium. Similarly, the use of general purpose processor cores in combination with smaller DSP or DSP-like cores is not new; and among the earlier examples of this design approach were the Texas Instruments TI 320C80 multiprocessor DSP and Cradle Technologies CT3400 DSP architecture.

5.3.1 Cell BE Programmability

Programmability is a critical factor in the widespread adoption and eventual success of a new architecture such as the Cell BE. In the previous sections, we described the nontraditional microarchitecture of the Cell BE and its differences from traditional CMP architectures. While the unique design of the Cell BE can potentially yield very high performance for suitable task or data-parallel applications, this performance comes at a price. The novelty of certain Cell BE technologies require the use of distinctly different programming models than those of traditional CMP format that most recent microprocessors follow. The programming models of traditional CMP architectures are essentially identical to parallel programming models that have been used for larger shared memory SMP (symmetric multiprocessing) machines before. These models and programming

methodologies, such as OpenMP and *pthreads*, had matured over time and became familiar technologies for programmers. Programmers with experience in such parallel programming tools and techniques could be able to port their applications to the Cell BE by using the processor's features to emulate message-passing or shared memory programming paradigms:Provided that the data partitioning can be done carefully to ensure that the data blocks fit in the SPE local stores, a conventional shared memory programming model can be approximated on the Cell BE by replacing shared memory accesses with DMA transfers. This method essentially uses the SPE local stores as *software-managed caches*. Alternatively, programmers can simulate a message passing model by utilizing the mailbox mechanism or DMA channels to exchange messages/data between the SPEs and the PPE.

While successfully porting applications to the Cell BE is possible using these methods, the task is significantly complicated due to the programming challenges that the Cell BE architecture poses. We briefly describe some of these challenges below.

- **Code partitioning for PPE and SPEs:** The heterogeneous multi-ISA architecture of the Cell BE implies that the PPE and SPE portions of the program executable should be compiled separately. This requires the programmer to manually partition the code into PPE and SPE portions; which might be a very time-consuming task both during the design of new Cell BE programs, and porting legacy applications to Cell BE.

- **Lack of "real" shared memory semantics:** The eight SPEs in the Cell BE have direct access to a very fast local memory and an even faster register file that is larger

than most traditional architectures, but the Cell BE lacks real shared memory that can be directly addressed and accessed by the SPEs. Until Cell BE compiler technology matures to a level of sophistication needed to automatically insert instructions for data movement between the main memory and the local stores, orchestrating this data movement remains the responsibility of the Cell BE programmer. Recent work by IBM [32] on this subject looks particularly promising.

- **Limited SPE local store size:** The SPE local stores have only 256KB of storage space for both code and data combined. This local store size is probably sufficient for the initial applications of the Cell BE architecture in multimedia and gaming: such applications generally include small, computationally intensive kernels of code that carry out tasks like sound effects generation or rendering scenes. However, many Cell BE SPE executables will not fit into the local stores while allowing a comfortable space for data and buffers for data movement operations, which can be critical for overlapping DMA operations with computation to improve performance. To handle executables that are too large to coexist with data in the local stores, the Cell BE programming libraries provide an overlay mechanism that could be used to transparently overlay the code portion of the local store with a new partitions of code during the execution of SPE programs. It will be the Cell BE programmer's responsibility to use mechanisms such as overlays or others to ensure that the SPE programs can fit and operate within the SPE local stores.

While advances in compiler technology([33],[32]) will undoubtedly make it easier to program processors like Cell BE in the future, most Cell BE porting projects such

as the HMMER work outlined in this paper currently require considerable programming effort. Cell BE architects have been aware of these programmability challenges from the start, and they proposed a multitude of programming models to guide Cell BE developers. Some of these models describe different methods of code partitioning between the PPE and SPEs, such as those outlined by Kahle et al. in [48]. Other Cell BE programming models emphasize task distribution strategies [60]. Some of these models are:

- Function offload model: This model allows the programmer to quickly boost the performance of an application by porting its most computationally intensive functions to the Cell BE SPE environment. The performance-critical functions could ideally be modified to take advantage of the SPE features, optimized and mapped to one or multiple SPEs. The rest of the program requires no change with the exception of the addition of a small stub which handles data transfer to the SPE and calls the SPE function as necessary.

- Computational acceleration model: In this model, most of the computationally intensive logic is parallelized and ported to the SPEs, and the PPE is relegated to the task of controlling the SPEs only. This approach requires more extensive modification and parallelization of the original code, and offers the potential of higher performance increases in return.

- Streaming models: Streaming models allow the developer to utilize SPE features such as fast local store access, large register files and low DMA latency by running small and fast computational kernels on the SPEs and streaming small sets of data to them. The SPEs can be organized as a pipeline, in which every SPE could be

running a different kernel. The PPE can then control the “stream processors”(SPEs) and the overall flow of data. Some applications(e.g. image processing) are already very viable candidates for streaming models of computation, and can benefit greatly from such implementations on the Cell BE.

5.4 HMMER on the Cell Broadband Engine

In order to take advantage of the novel heterogenous multiprocessor architecture of the Cell Broadband Engine, application code needs to be modified to utilize the eight SPEs and the associated local stores. This currently requires application programmers to manually partition the application code into parts to be executed on the PPE and the SPEs, in addition to devising methods to utilize the DMA channels efficiently to shuttle data to and from the SPEs. Future Cell BE compilers will make it possible to automate this partitioning process, and [32] describes ongoing work in this area.

When used to compare protein motifs against protein sequences, HMMER can be used in one-to-one,one-to-many,many-to-many or many-to-one search configurations using different applications in the suite. The *hmmsearch* application, which we studied on our earlier work as part of the BioBench suite, searches a sequence database against a single HMM; and the *hmmpfam* application searches a database of HMMs against a single sequence. While both applications rely heavily on the Viterbi algorithm and are largely similar, they have different characteristics and require different parallelization strategies. The importance of the programs in the HMMER suite and their high computational cost prompted many researchers to study and attempt to improve their performance using dif-

ferent approaches. Some of the earlier work on improving HMMER performance targeted *hmmsearch* using strategies like FPGA hardware acceleration[6], use of SIMD instruction sets[96], and the use of unconventional architectures such as GPU clusters[44] or network processors[101].

We chose to implement the *hmmsearch* application on the Cell BE architecture. *Hmmsearch* reads sequences from a database one by one and executes the Viterbi algorithm on each sequence using a common HMM. If the score of a sequence is higher than a predefined threshold, the most likely path for the HMM is computed using the traceback algorithm; and the best hits are recorded in a report. This structure lends itself well to parallelization:shared-memory parallel implementations of *hmmsearch* typically distribute sequences to different threads; each of which proceed to execute the Viterbi algorithm. No inter-thread communication is necessary since each sequence is a separate entity which can be processed independently of the others. As a result, a high degree of parallelism is possible with fairly basic synchronization, resulting in good scalability.

A typical Cell BE porting effort would begin with determining the most computationally intensive functions of the workload that can be parallelized and executed on the SPEs. Based on our earlier detailed analysis of the HMMER profile searching workload, the Viterbi algorithm was the function most suitable for implementation on the Cell BE SPE.

5.4.1 Code Partitioning

We used the standard HMMER distribution as the starting point for Cell-HMMER. Code for basic tasks like reading FASTA-formatted protein sequence files and HMM file manipulation are directly ported from HMMER and run on the PPE.

To partition the main functionality of HMMER between the Cell BE and the SPE, we referred to the literature on established parallel programming *design patterns* in addition to the Cell BE programming models mentioned earlier. The term "design pattern" is used to describe generalized solutions to problems that commonly occur in software design, and a wide variety of design patterns are available for many different application types(The influential book by Gamma et al.[40] contains a wide selection of such patterns and provides a good introduction to the concept). We found many of the patterns from the parallel program design patterns literature to be readily applicable to the problem of porting applications to the Cell BE. Cell-HMMER was implemented using a variation of the "Manager-Workers" parallel design pattern that follows the Cell BE computational acceleration model described earlier. Ortega-Arjona et al. describe the details of the Manager-Workers parallel design pattern as well as other patterns in [74].

The Manager-Workers pattern provides a particularly suitable programming template for applications with the following characteristics:

- The program needs to process data organized as a large number of independent work units.
- The order of the input data needs to be preserved.
- There are no constraints on how the data is distributed to the processors.

In Manager-Workers, one *manager* thread is tasked with creating the *worker* threads, distributing work units to them and gathering results as the units are consumed by the workers. The *worker* threads are tasked with requesting and receiving data from the manager, carry out the necessary computation on this data, and return results to the manager. In this pattern, no communications between workers are allowed. The only communication takes place between the manager and the workers. The manager distributes the work units to the workers as they request them. For a problem setting with a large number of work units with varying lengths, this strategy provides a "natural load balance"[74]. While some worker threads are busy processing large work units, some others can process several smaller units; balancing the load on the overall system.

HMMER needs to compute the Viterbi algorithm on a large number of protein sequences with differing sizes, all of which can be processed independently. We believe this characteristic of HMMER makes it a good candidate for using the Manager-Workers pattern. The *manager* role is assigned to the PPE (which will be dedicated to this task only), and each SPE is a *worker*. There is no communication between the SPEs, and the communication between the PPE and the SPEs is done using low-latency synchronization primitives such as atomic variables and mutexes. The *manager* PPE component, and the *worker* SPE components will be described in the next section. Figure 5.9(adapted from [74]) illustrates the overall communication and data flow in Cell-HMMER.

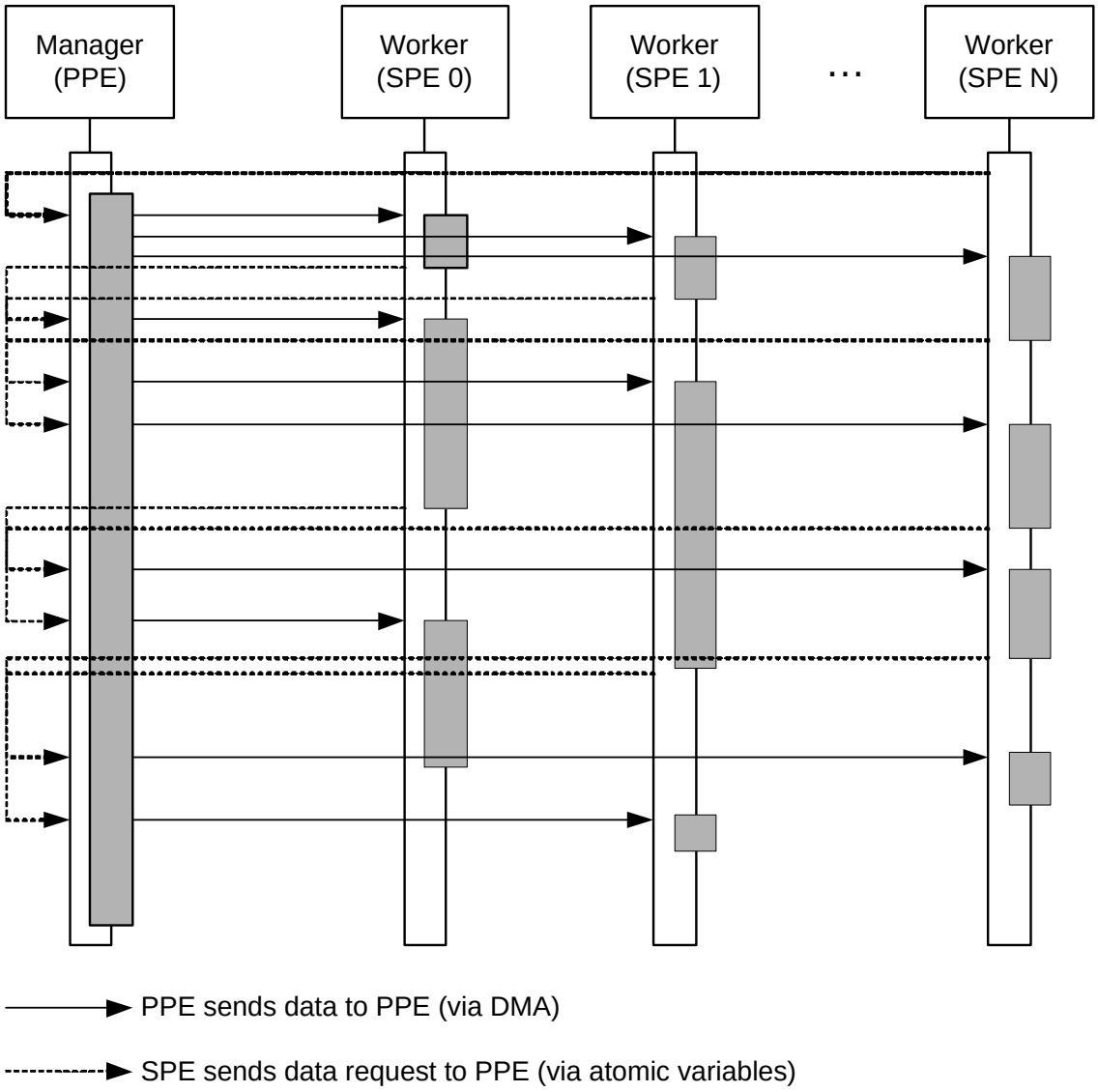

Figure 5.9: Cell-HMMER communication and data flow

5.4.2 SPE Component

The SPE component of Cell-HMMER is an SPE program which initiates a DMA transfer to copy the HMM, and enters a loop to process protein sequences transferred from the main memory. Within this loop, the SPE component requests DMA transfers of sequences, computes the Viterbi result, and transfers the results back to the main memory. The most important task of the SPE program is running the Viterbi algorithm.

The information to transfer the HMM data structure, such as the memory address and length of the HMM, are conveyed to the SPE threads as part of the initial context, a 128-byte data structure passed to each SPE when the SPE threads are being started. This allows the SPE threads to start the DMA transfers for the HMM data without help from the PPE. The HMM is almost always the largest single DMA transfer in the application and need to be transferred only once, therefore overlapping this transfer with the PPE's setup and creation of the initial sequence buffer improves overall performance. With the HMM transfer complete, the SPEs can now proceed to request sequences and execute the Viterbi algorithm.

The most important constraint for implementing the Viterbi algorithm on the Cell BE is the storage requirements of this algorithm. We previously showed that the Viterbi algorithm on a Plan7 HMM has a memory footprint in the order $O(N.M)$ if we need all the intermediate probabilities at the end of the operation. For a typical HMM of length 200, this implies that only sequences less than 1250 characters could be processed without leaving any space in the local store for the program binary, the HMM itself and other variables that might be needed by the program. This situation requires the use of a differ-

ent approach to implementing the Viterbi algorithm in the limited SPE environment. One possible approach is converting the Viterbi algorithm to a streaming workload as done in[44], another might be changing the Viterbi algorithm to process its inputs in smaller units and combine results later. Either one of these approaches would require nontrivial modifications to the Viterbi algorithm.

The Viterbi algorithm operates by evaluating each HMM state in a loop that iterates through all the symbols in the input sequence. Each iteration of this sequence loop is dependent on the values produced by the previous iteration. However, the algorithm requires all of the previous values produced by each iteration for a full traceback. A key observation that made the Cell-HMMER work possible was that we only need *complete* traceback information on a few sequences which result in high enough Viterbi scores to be considered significant hits. A great majority of sequences do not have such high scores; and therefore will never need a full traceback. If a full traceback is not necessary, the Viterbi algorithm could be modified to discard the intermediate probability values produced by all but the previous iteration at any point. For an HMM with N states and a sequence of M symbols; this modification effectively reduces the storage requirement of the Viterbi algorithm from $O(N.M)$ to $O(N)$, making it possible run the algorithm in the limited SPE local stores on the Cell BE. We wrote our SIMD Viterbi implementation using this approach, and allocated a 150KB buffer in each of the 256KB SPE local stores. This buffer size can accommodate a maximum HMM length of 500; which covers more than 93% of all the protein families in Pfam according to our analysis. Future Cell BE implementations with larger SPE local stores (which might be produced using promising dense cache technologies like eDRAM or zRAM) could lessen the impact of this limita-

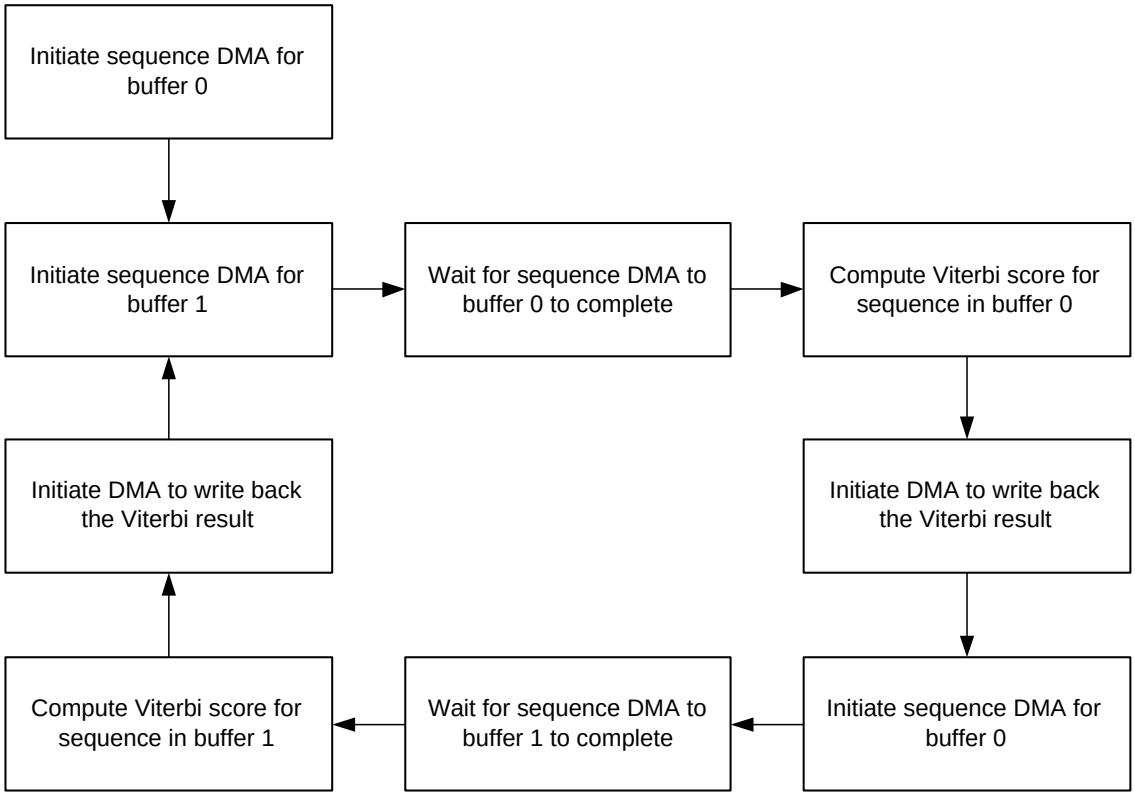

Figure 5.10: Double buffering as used by Cell-HMMER

tion. We are planning to develop a version of Cell-HMMER without HMM length limits as part of our future work on the subject.

In order to improve the performance of Cell-HMMER by overlapping DMA operations with computation, we use a “double buffering” technique as shown in Figure 5.10. This method uses two sequence data buffers in the SPE local store: Each SPE start the Viterbi processing by initiating a DMA to one of the buffers. As soon as this DMA is complete, the SPEs enter a loop where the SPE initiates a sequence DMA to the other buffer, and starts processing the sequence which was transferred as the result of the previous DMA. Since processing a sequence using the Viterbi algorithm generally takes longer than DMA’ing from the main memory to the SPE, this alternation between the two

```
for (i = 1; i <= L; i++) {
  mmx[i][0] = imx[i][0] = dmx[i][0] = -INFTY;
  for (k = 1; k <= hmm->M; k++) {
      /* match state */
    mmx[i][k]  = -INFTY;
      if   ((sc = mmx[i-1][k-1] + hmm->tsc[TMM][k-1]) > mmx[i][k])
    mmx[i][k] = sc;
      if   ((sc = imx[i-1][k-1] + hmm->tsc[TIM][k-1]) > mmx[i][k])
    mmx[i][k] = sc;
      if   ((sc = xmx[i-1][XMB] + hmm->bsc[k]) > mmx[i][k])
    mmx[i][k] = sc;
      if   ((sc = dmx[i-1][k-1] + hmm->tsc[TDM][k-1]) > mmx[i][k])
    mmx[i][k] = sc;
      if   (hmm->msc[dsq[i]][k] != -INFTY) mmx[i][k] += hmm->msc[dsq[i]][k];
      else
    mmx[i][k] = -INFTY;

      /* delete state */
   dmx[i][k] = -INFTY;
      if   ((sc = mmx[i][k-1] + hmm->tsc[TMD][k-1]) > dmx[i][k])
    dmx[i][k] = sc;
      if   ((sc = dmx[i][k-1] + hmm->tsc[TDD][k-1]) > dmx[i][k])
    dmx[i][k] = sc;

      /* insert state */
      if   (k < hmm->M) {
    imx[i][k] = -INFTY;
          if   ((sc = mmx[i-1][k] + hmm->tsc[TMI][k]) > imx[i][k])
              imx[i][k] = sc;
          if   ((sc = imx[i-1][k] + hmm->tsc[TII][k]) > imx[i][k])
              imx[i][k] = sc;
          if   (hmm->isc[dsq[i]][k] != -INFTY) imx[i][k] += hmm->isc[dsq[i]][k];
          else
              imx[i][k] = -INFTY;
  }
 }
}
```

MAX(incoming transitions to M_k)

MAX(incoming transitions to D_k)

MAX(incoming transitions to I_k)

Figure 5.11: Main loop of the Viterbi algorithm code in standard HMMER

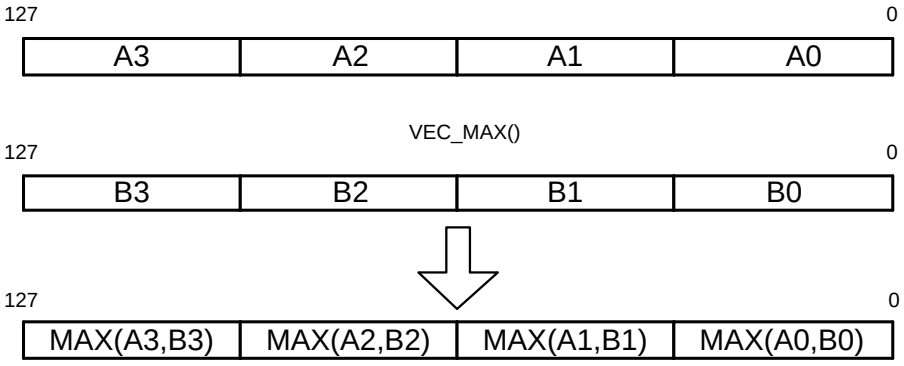

Figure 5.12: 128-bit SIMD vector maximum operation

buffers efficiently hides the DMA transfer time by completely overlapping computation with communication by alternating between the two buffers. The size of the two sequence buffers are configurable. Our analysis of the latest version of the SwissPROT protein sequence database, we found that only 4 of the more than 250,000 sequences in SwissPROT were longer than 10,000 characters. We decided to use a sequence buffer size of 10KB in our experiments, resulting in a total SPE local store usage of 20KB for sequence data.

The primary task of the SPE component of Cell-HMMER is high-performance

SIMD Viterbi computation. Such a SIMD Viterbi implementation exists in the form of the AltiVec implementation of Lindahl; and the similarities between the AltiVec and Cell BE SPE instruction sets and C/C++ programming intrinsics would have simplified the porting effort a great deal. However, this code implemented a full Viterbi algorithm with traceback; which is not possible on the limited local store of the SPE for most sequence and HMM sizes. For this reason, we chose to write a new SIMD implementation of the Viterbi algorithm for the Cell BE SPE to operate on a much more stringent memory footprint, keeping only the intermediate results for the previous pass in the inner loop. Our implementation is written using the SPE C programming intrinsics. It features a loop unrolled 4 times (as opposed to 8 times in the Lindahl implementation) and its design represents a compromise between readability and high performance.

The use of SIMD vector operations is critical to obtain the highest possible performance on the Cell BE SPE; and the performance advantage of Cell BE in many different workloads can be directly attributed to its efficient SIMD instruction set among its many architectural features. Similarly, SIMD implementations of HMMER are crucial to obtain the highest HMMER performance on many general microprocessor architectures. The IBM/Freescale PowerPC architecture can execute HMMER remarkably faster than other architectures largely due to the existence of a very efficient AltiVec SIMD implementation of HMMER by Lindahl. The x86 microarchitecture have been updated with a wide variety of SIMD instructions over the last ten years, but SIMD implementations of HMMER on the x86 platform still can not perform as fast as the PowerPC versions on a clock-by-clock adjusted basis. The reasons for this performance difference lie in the instruction sets, and provide useful insights about HMMER performance on the Cell BE

as well as other platforms.

In order to simplify the Viterbi algorithm and improve its performance, HMMER converts and stores the HMM data in log-odds format, where all transition probabilities are converted into negative logarithms in order to change the multiplication operations into simple additions. The original Viterbi algorithm requires finding the minimum among the results of these multiplication operations, and the change to negative logarithms converts the minimum operator to a maximum operator. This part of the code is repeated for each state of the HMM and for each symbol in the sequence, and its sheer frequency renders it very important for high HMMER performance. A portion of this code is shown in Figure 5.11.

In a straightforward SIMD implementation of this code, the addition operations and the subsequent comparisons to find the maximum value for each state group can be replaced with vector operations. Assuming that the 128-bit vector registers in many SIMD architectures could be used to store 4 32-bit unsigned integers representing the intermediate values, this could be greatly facilitated by a 4-way vector maximum operation which can compare two such vector registers. The AltiVec SIMD instruction set used by Lindahl's high-performance HMMER implementation has a single instruction (VMAXUW) which accepts two 128-bit registers holding four 32-bit unsigned integers each and places the larger integer in each slot in the corresponding slot in the destination register. While the VMAXUW instruction is supported by the Cell BE PPE, no equivalent single instruction exists for the Cell BE SPE. The same operation can instead be done by using two instructions in the Cell BE SPE ISA: The CLGT d,a,b instruction sets the bits of the destination register to 1 if the integer in the corresponding vector slot in register a is greater

AltiVec /VMX (Cell BE PPE)	SSE2 (x86)	Cell BE SPE
VMAXUW d,a,b	MOVDQA xmm3,xmm2 PCMPGTD xmm3,xmm1 PAND xmm2,xmm3 PANDN xmm3,xmm1 POR xmm3,xmm2	CLGT d,a,b SELB c,b,a,d

Figure 5.13: Instructions needed for vector maximum

than that in register b. A following SELB instruction then is used to select the vector elements and place them in the destination register.

The Intel SSE2/SSE3 SIMD extensions do not provide a vector maximum operation instruction for 32-bit unsigned integers(“doublewords” in x86 terminology) even though they include similar instructions for 8-bit unsigned (PMAXUB) and 16-bit (PMAXSW) signed integers. Without such an instruction, the best implementation of a vector maximum operation of the type used in the HMMER Viterbi code requires the use of 5 instructions. The lack of a 32-bit vector maximum instruction for signed integers seems to have a fairly large negative impact on the HMMER performance of the x86 architecture. Walters et al. [96] identify and discuss this issue in their description of the SSE2/SSE3 version of the HMMER Viterbi code they implemented.

Cell-HMMER uses both the PPE and SPE for running the Viterbi algorithm, and the higher performance of the 32-bit unsigned integer vector maximum operation on the Cell BE PPE and SPEs had a very important impact on our results. The SIMD instruction sequences required to implement the unsigned integer vector maximum operation on the x86(SSE2), PowerPC(AltiVec) and the Cell BE architectures are shown in Figure 5.13.

In late 2006, Intel announced that a 32-bit unsigned integer vector maximum instruction(PMAXUD) will be part of the SSE4 SIMD extension ISA to be used in future x86 instructions[7]. We believe that future HMMER implementations that use the PMAXUD instruction could demonstrate much higher performance on the x86 architecture.

The end of the sequence database is communicated to the SPE threads by a special token flag variable set by the PPE thread. The SPE threads check this flag every time the sequence buffer is empty. If the flag is set, the SPE threads complete their operation, transfer their last Viterbi results to the main memory via DMA and exit.

5.4.3 PPE Component

The PowerPC-compliant PPE in the Cell BE is an in-order, 2-way simultaneous multithreading processor. In Cell-HMMER, we primarily use the PPE as a task controller and an I/O processor for HMM and sequence data. While the PPE is a powerful processor in its own right and could have been used to speed up Viterbi processing, the only place we use the PPE for computation is during the processing of sequences which require traceback. The reasons behind this design choice have to do with the demands of the sequence input and synchronization:

- **Sequence input takes time:**Reading every sequence from a large sequence database file, memory allocation and manipulation of the data structures used to represent sequence entries is a time-consuming task; and this step needs to be overlapped with Viterbi computation on the SPEs for maximum performance. In many test cases, we found that the time required to read and set up all sequences in a sequence

database was very close to the time required to process these sequences on 8 or more SPEs. This keeps the PPE busy throughout the buffering process, and does not leave much time for computation. In fact, sequence input becomes a limiting factor on 2-processor Cell BE blade servers, which make up to 16 SPEs available for Cell BE programs. We found that file input became a bottleneck with larger SPE numbers, as we will discuss later.

- **Orchestrating multiple SPEs requires complex synchronization:**With multiple SPEs requesting sequences from the buffer and sending results back, the synchronization task requires the resources of the PPE. The PPE constantly monitors the state of the sequence buffer, stops and restarts SPE threads as necessary, and gathers results at the end. We believe implementing a completely self-arbitrating HMMER port would have been much more difficult and probably have not resulted in much higher performance.

The Cell-HMMER PPE application starts by reading the HMM file, converting into negative logarithm format, and packing the HMM into an efficient data structure that can be transferred to the HMM buffer of the SPEs in as few DMA operations as possible. The PPE then starts reading sequences from the sequence database into a buffer in the main memory, setting up a data structure called a *"job entity"* for each sequence as it goes. Each sequence is stored in a 128-byte aligned memory address to facilitate SPE-initiated DMA transfers, and each job entity contains information such as the sequence length, sequence name and the address of the sequence in the memory. In order to ensure that the SPEs can not catch up to the PPE and deplete the buffer quickly, the PPE buffers a

configurable number of sequences before starting the SPE threads.

Buffer synchronization is done using atomic variables in the main memory, which can be a atomically read and incremented/decremented by the SPEs and the PPE. As soon as the SPE threads start up and get ready to start processing sequences, they request the next available job entity entry by issuing a DMA request from the PPE buffer. This data structure contains all the information the SPE requires to initiate a DMA transfer and start processing, and the rest of the sequence transfer operation can proceed without help from the DMA. The SPEs decrement the number of sequences in the buffer as they consume sequences, and the PPE thread increments it as it reads more sequences and adds to the buffer. When the buffer is empty, the SPE threads will go to sleep until the PPE can replenish the buffer with more sequences and wakes them up. This continues until all the sequences in the sequence database are read.

The initial sequence buffer should ideally be large enough to ensure temporal separation between the PPEs and the SPEs, so that the SPEs can not catch up with the PPEs very quickly. A second reason for this separation is the performance hit incurred by the SPE-initiated DMA transfers when the data comes from the L2 cache instead of the system memory. As a result, the aforementioned temporal separation could also contribute to performance by increasing the probability that the cache lines containing the sequences are evicted from the Cell BE cache by the time they are needed by the SPEs. On the other hand, the buffer should not be so large since the SPEs can not do useful work while the sequences are being buffered in the very beginning of the application. We tuned the initial buffer size by conducting experiments, and varied it according to the number of sequences in the sequence database. For most of our experiments, the size of this buffer

was in the range of 2 to 3 percent of the overall number of the sequences in the input database.

The Cell BE architecture supports multiple page sizes. 4KB pages can be used simultaneously with any two of 64KB, 1MB or 16MB pages. We used 16MB memory pages to allocate sequence data buffers in order to minimize the negative impact of frequent TLB reloads.

A naive and less complex implementation might possibly read all sequences into a buffer, and then start the SPE sequences. The SPEs could then self-arbitrate without any help from the PPE thread, and process the sequences in the buffer. Reading all sequences from a large database can take a long time, and the performance of such an implementation will clearly suffer since the SPEs are doing nothing during this time. In our implementation, the relatively complex PPE function to read and simultaneously distribute sequences to the SPEs keeps the PPE busy until all the sequences are read and buffered. As a result, overlapping Viterbi computation on the SPEs with sequence input on the PPE improves the performance of Cell-HMMER significantly.

After the SPEs process the sequences, the results of the Viterbi algorithm are transferred to the main memory by the SPEs via DMA without any help from the PPE thread. and the PPE compares each result with a threshold value to determine whether a PPE Viterbi traceback run is necessary. Traceback computation can not be done on the SPEs because of the limited size of the SPE local stores. Running the Viterbi algorithm with full traceback is not a problem on the PPE, as the PPE can access large amounts of main memory directly. For sequences whose scores exceed the threshold, the PPE repeats the Viterbi computation with full traceback information for postprocessing. We use the very

fast AltiVec(VMX) implementation by Lindahl for PPE-side Viterbi computations. As stated earlier,only a very small number of sequences actually need traceback: among the 5 HMMs that we used for our experiments(Figure 5.14), the highest number of significant hits against the more than 250,000 sequences was only 148. We observed that typically less than 0.1 percent of sequences require traceback operations in large HMMER runs using common threshold values. Consequently, the PPE-side Viterbi processing of has little negative impact on the performance of Cell-HMMER for many typical HMMER jobs. For this reason, we did not need to utilize the two-way SMT feature of the PPE to improve the performance of this step. Figure 5.14 shows the number of tracebacks (significant hits) for each of the five HMMs used in our experiments when searched against the SwissPROT database.

Finally, the PPE could be used for postprocessing of traceback information to display alignment statistics and a histogram to the user. The contribution of this step to the overall execution time of HMMER is insignificant in comparison to the Viterbi algorithm, and we did not implement this part in the first version of Cell-HMMER. The code for this step does not require a different Cell BE implementation and could be adapted from a standard HMMER distribution with little effort.

5.5 Experiments

5.5.1 Experimental Methodology

Cell-HMMER was developed in the C programming language using the public version 1.0 of the Cell BE Software Development Kit (SDK) by STI. The basis for sequence

HMM Name	Length (states)	Description
COQ7	100	Ubiquinone biosynthesis protein COQ7
Maf1	200	Maf1 regulator
Lipoprotein_1	300	Borrelia lipoprotein
APG9	400	Autophagy protein Apg9
GerA	500	Bacillus/Clostridium GerA spore germination protein

Figure 5.14: HMMs used in the experiments

and HMM input/file manipulation code and the AltiVec Viterbi implementation was the HMMER 2.3.2 by Eddy[31].

The development of Cell-HMMER was started before there was any Cell BE hardware available to us, and we used the Mambo Cell BE simulation framework from IBM[78] for our initial development work. Mambo is a very flexible cycle-accurate simulator which faithfully models both SPE and PPE pipelines and DMA effects, and using Mambo significantly simplified our work. Later stages of the development were done on both Mambo and a Sony PlayStation 3 gaming console running the Linux operating system.

The input data we used for our experiments came from well-known bioinformatics databases: Pfam[17] and SwissPROT. We chose five different HMMs that range between 100 and 500 states from Pfam; and these HMMs are listed in Figure 5.14. As mentioned earlier, the maximum HMM length usable on Cell-HMMER is 500; which covers 93 percent of HMMs in Pfam. Allocating the maximum amount possible to the HMM buffer required us to use a conservative setting of 10000 for maximum sequence size, and we processed the complete SwissPROT sequence database to strip away sequences longer

Processor	Clock Speed	L1 Cache	L2 Cache	RAM	# of Cores
Intel Pentium 4	2.8 GHz	16 KB	1 MB	2 GB	1
AMD Opteron 246	2.0 GHz	128 KB	1 MB	4 GB	1
AMD Opteron 280	2.4 GHz	256 KB	2 MB	4 GB	2
AMD Opteron 280 (2-way SMP)	2.4 GHz	256 KB	2 MB	4 GB	4

Figure 5.15: Configurations of the x86 systems used in the experiments

than 10000 characters (Of the more than 250,000 sequences in SwissPROT, only 4 were longer than this limit.).

We obtained performance measurements on real Cell BE hardware by running Cell-HMMER on a prototype dual-processor Cell server blade at IBM T.J.Watson Research Center.The two 3.2GHz Cell BE processors on this system make up to 16 SPEs available to the user, however we used only 8 SPEs for the relatively small SwissPROT data set. To compare the HMMER performance of Cell BE with x86 generations from the last two generations, we obtained performance measurements on four different x86 system configurations. The details of these systems are given in Figure 5.15.

For the baseline x86 tests, we used the standard HMMER 2.3.2 source code distribution with *pthreads* multithreading support. Since Cell-HMMER currently does not have the final postprocessing step, we commented out the postprocessing functionality before compiling HMMER on each target system using the *gcc* C compiler at the optimization level $-O2$. As the standard HMMER distribution does not use SSE2/SSE3 SIMD acceleration, the x86 code does not have SIMD support. It should be noted that versions of x86 HMMER with SSE2/SSE3 acceleration exist[96]. The lack of source code for these versions would have meant that the postprocessing functionality could not be commented

Architecture	Cores	Frequency	COQ7 Time (s)	MAF1 Time(s)	Lipoprotein Time(s)	APG9 Time(s)	GerA Time(s)
Intel Pentium 4	1	2.8 GHz	344.66	681.91	1055.63	1371.09	1708.05
AMD Opteron 246	1	2.0 GHz	423.7	825.9	1389.17	1810.1	2239.09
AMD Opteron 280	2	2.4 GHz	169.09	330.93	509.13	661.35	815.33
AMD Opteron 280 (2-way SMP)	4	2.4 GHz	86.13	174.32	288.86	332.68	411.9
Cell BE - 1 SPE	1	3.2 GHz	97	115.44	135.4	155.83	173.68
Cell BE - 2 SPEs	2	3.2 GHz	48.63	57.81	67.75	77	86.95
Cell BE - 3 SPEs	3	3.2 GHz	32.47	38.6	45.22	51.41	58.06
Cell BE - 4 SPEs	4	3.2 GHz	24.36	28.98	33.99	38.61	43.6
Cell BE - 5 SPEs	5	3.2 GHz	19.57	23.18	27.2	30.96	34.89
Cell BE - 6 SPEs	6	3.2 GHz	16.32	19.35	22.68	25.78	29.1
Cell BE - 7 SPEs	7	3.2 GHz	14.03	16.6	19.48	22.17	24.98
Cell BE - 8 SPEs	8	3.2 GHz	21.9	14.57	17.07	19.41	21.85

Figure 5.16: Execution times for the test HMMs

HMM	Speedup vs. P4	Speedup vs. Opteron 246	Speedup vs. Opteron 280	Speedup vs. Opteron 280(2P)
COQ7	15.73x	19.34x	7.72x	3.93x
Maf1	46.80x	56.68x	22.71x	11.96x
Lipoprotein_1	61.84x	81.38x	29.82x	16.92x
APG9	70.63x	93.25x	34.07x	17.13x
GerA	78.12x	102.47x	37.31x	18.85x

HMM	Normalized Speedup vs. P4	Normalized Speedup vs. Opteron 246	Normalized Speedup vs. Opteron 280	Normalized Speedup vs. Opteron 280(2P)
COQ7	13.77x	12.09x	5.79x	2.94x
Maf1	40.95x	35.42x	17.03x	8.97x
Lipoprotein_1	54.11x	50.86x	22.37x	12.69x
APG9	61.80x	81.59x	25.55x	12.85x
GerA	68.40x	64.04x	27.98x	14.13x

Figure 5.17: Speedup obtained by Cell-HMMER using 8 SPEs

out, therefore we chose not to use these in our experiments.

5.5.2 Results

Figure 5.16 shows execution times of HMM searches for our 5 test HMMs on different platforms and the Cell BE. We used default HMMER options and searched the complete SwissPROT sequence database (250,000 sequences) against the test HMMs. For the Cell BE, we repeated the experiment while varying the number of active SPEs from 1 to 8. The graphs in Figure 5.18 compare the performance of all twelve configu-

rations for different input sizes. The performance data clearly shows Cell-HMMER has significant performance advantages over conventional architectures, even with only one SPE active in many cases. While the performance advantage in the single SPE can be largely attributed to the use of SIMD optimizations coupled with the fast local store; the performance gap widens as number of active SPEs increase and the parallelism inherent in the workload can be fully exploited.

Using all 8 SPEs, Cell-HMMER on a 3.2GHz Cell BE not only performs up to 37 times faster than a dual-core Opteron at 2.4GHz; but also easily outperforms a 2-way SMP configuration using the same Opteron processors. The clock-adjusted 14.13x performance differential between the Cell BE and the 4-core SMP setup suggests that Cell-HMMER is likely to remain competitive against next-generation quad-core x86 processors, even factoring in an optimistic 50% performance improvement for the future x86 part. The introduction of 4-way vector maximum instruction in SSE4 might change the situation in the x86 architecture's favor: such an instruction will make it possible to improve the performance of x86 HMMER performance significantly(The results in [44] suggest that Lindahl's SIMD implementation of the Viterbi algorithm code provides a 4x-7x performance advantage to the PowerPC architecture, which has such an instruction).

For our relatively small input data set, we observe almost perfectly linear scaling on the Cell BE as the number of SPEs increase. The decrease in execution time was linear for all but one HMM: the smallest data set, COQ7. The execution time for this input increased as the number of SPEs increased from 7 to 8. This is almost certainly due to the additional overhead introduced when the sequence buffer gets depleted too frequently. Since the input HMM was fairly small and the SPEs were able to complete

the Viterbi algorithm very quickly, the increased contention for sequences harmed performance. Our experiments using up to 16 SPEs on the 2-processor Cell BE blade servers at IBM confirmed that the same problem manifests itself at different points for other data sets: Inevitably the SPEs catch up to the PPE, and the PPE frequently has to stop the threads and replenish the buffer. Our default buffer size (2000) was too small to reduce the occurrence of these events, and we are considering using a tunable buffer to overcome this scalability problem. Such a buffer can either adjust its size statically using a formula based on the size of the input HMM and the number of active SPEs; or adaptively change its size during execution if the SPEs start catching up the PPE too frequently.

Figure 5.17 illustrates the speedup of HMMER searches on the Cell BE with 8 SPEs over other systems in our tests. When the numbers are normalized by the clock speed to eliminate the effects of differences in clock speeds; the Cell BE architecture and Cell-HMMER outperforms all the other systems by more than an order of magnitude in many cases.

5.6 Related Work

The computational complexity of bioinformatics algorithms and the resulting performance issues have been recognized fairly early on, and numerous attempts have been made to accelerate important bioinformatics applications and workloads. Many of these attempts were straightforward optimization and parallelization work; while some followed unconventional approaches.

Some of the earliest attempts to improve the performance of computationally de-

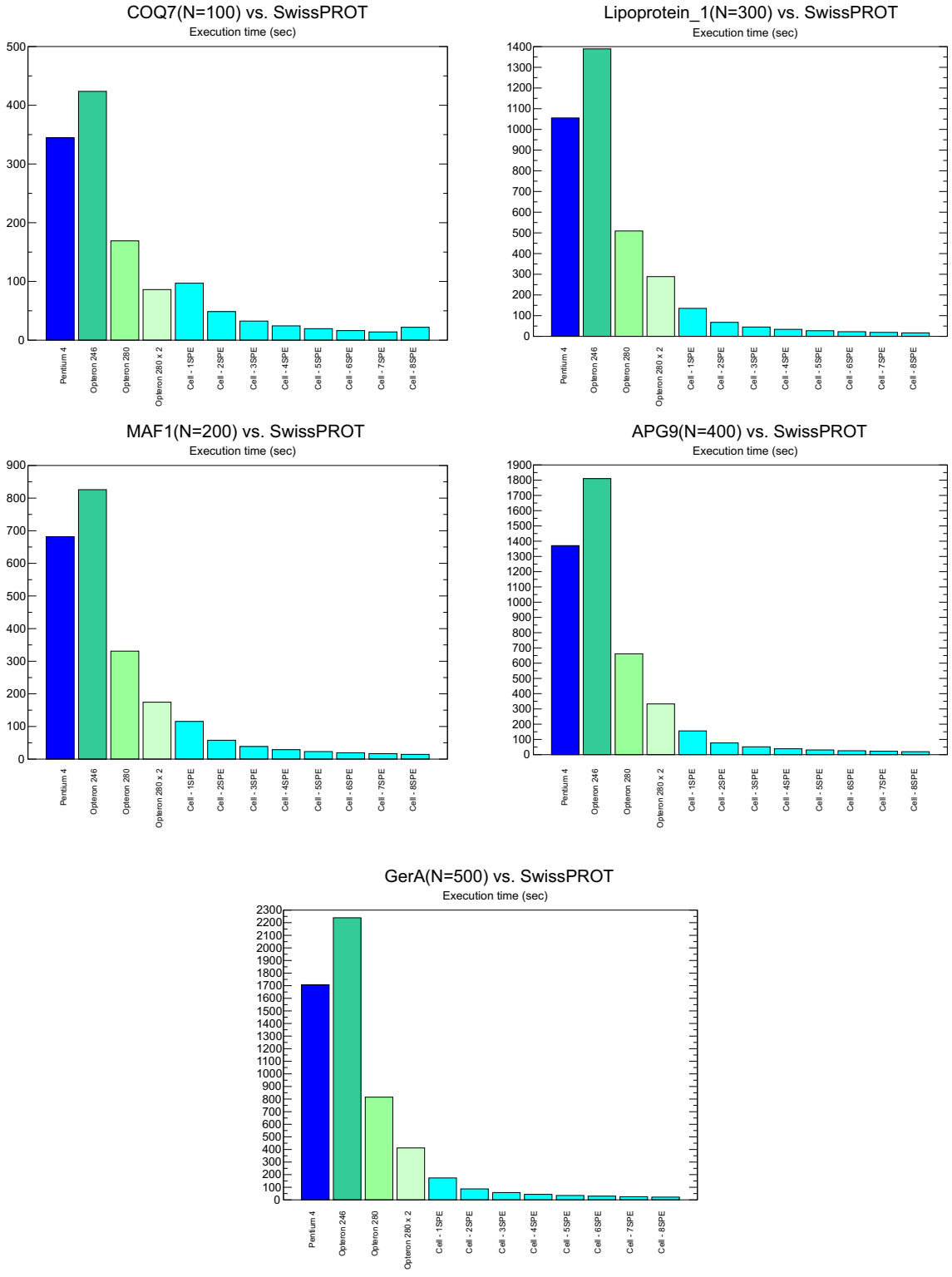

Figure 5.18: Performance data

manding bioinformatics workloads involved the use of fast, heuristics-based algorithms to replace computationally intensive, exact algorithms. Perhaps the best known examples of this approach are the BLAST[12] and FASTA[] algorithms which were devised to approximate the Smith-Waterman algorithm [85]).

Current approaches to accelerating bioinformatics workloads range from special-purpose hardware to optimized versions of the original applications; and many solutions that have been proposed fall somewhere in between. Hardware acceleration of bioinformatics applications offers great improvements in performance at the expense of decreased flexibility. The special-purpose hardware approach requires the mapping of computationally intensive parts of the underlying algorithm to an ASIC or more typically an FPGA implementation. These ASICs or FPGAs are then installed on a host system which handles high-level I/O to access the database, or execute non-critical sections. Such implementations [22],[72],[65],[6] have usually been successful in many niches where a single, specific algorithm is targeted or the targeted class of applications share common operations that can be sped up by the acceleration hardware.

Bioinformatics applications have also been accelerated on non-conventional architectures such as commodity graphics processing units (GPUs) [44], multicore digital signal processors (DSPs) [63], and network processors [101] with some degree of success. In many cases, such application specific hardware is used to close the performance gap between exact algorithms (which yield more precise results) and their fast approximations.

The software optimization approach involves rewriting or modifying a particular bioinformatics application by either using a more efficient algorithm, by making use of

inherent parallelism, or by utilizing mechanisms used in recent microprocessors to accelerate other kinds of applications. A very common example is the use of SIMD extensions that were originally designed to address the needs of multimedia applications [82]. Many bioinformatics applications employ algorithms that can benefit from the use of SIMD extensions, and in many cases the best performing versions of these applications depend on SIMD instructions for high performance.

Several papers in the computer architecture and performance evaluation literature have recognized HMMer as an interesting and computationally intensive workload even before its inclusion in the SPEC 06 benchmark suite. Our earlier work describing the BioBench suite[11] was probably among the first workload characterization studies of HMMER. Later studies by Li et al.[57] and Bader et al.[14][13] also characterize HMMER as part of their proposed benchmark suites. The results presented in these studies mostly coincide with our findings, such as: many bioinformatics benchmarks (particularly those in the domain of genomics) have little or no usage of floating point instructions; a higher degree of ILP is available in bioinformatics applications than in SPEC INT or FP benchmarks; and branches in bioinformatics applications are slightly less predictable than the SPEC benchmarks used in the studies. The few discrepancies between the results of these studies can probably be attributed to differences in the input data sets and methodologies.

Some recent work on HMMer performance optimization concentrated on using conventional processor architectures. Walters et al.[96] an MPI version of HMMer for cluster systems, and presented results suggesting better scalability and performance than the freely available PVM-based HMMer. They also presented a version of HMMer which

uses the Intel SSE2 128-bit SIMD extensions to accelerate the Viterbi algorithm, much like Lindahl's work involving an AltiVec port of HMMer. Their SSE2-optimized HMMer port exhibits a 23.2 percent reduction in execution time. Combining their SSE2 optimizations with their MPI port of HMMer, Walters et al. were able to obtain a speedup of 6.65 times using an 8-processor cluster. This implementation is most likely the best performing HMMer implementation for x86-based PC clusters. In a similar study, Landman et al.[55] describe a version of HMMer optimized for the AMD Opteron architecture[49] which makes use of loop optimizations and conditional move instructions for hard-to-predict branches. Their HMMer implementation runs up to 1.96 times faster than stock HMMer on the same hardware, despite not using SSE2 SIMD instructions.

Two recent studies that studied HMMer on unconventional architectures deserve special mention. Wun et al.[101] presented an implementation of HMMer on the Intel IXP 2850 network processor. Their implementation, JackHMMer, uses a different approach than ours for exploiting the coarse-grained parallelism of HMMer. JackHMMer utilizes a different HMMer operation model which compares a HMM model database against a single sequence, as opposed to the more common model of comparing a single HMM model with many sequences. In JackHMMer, HMM data is kept in the faster SRAM-based memory while intermediate data resides in the DRAM main memory. The sequence information is stored in the local scratchpad memory of the IXP MEs (micro-engines, i.e. execution units). The sections of model and intermediate data required for the multiple jobs running on the MEs are laid out in the form of "work packets", which are then transferred to SRAM from DRAM by the MEs. This data arrangement is feasible on the Intel IXP architecture due to the existence of real shared memory, and the

relatively large size of SRAM compared to the 256KB SPE local store size. The authors' implementation of HMMER running on a 1.4GHz IXP network processor is 1.82 times faster than an optimized HMMER implementation running on a 2.6GHz Intel Pentium 4 processor. However, the optimized x86 version of HMMER used in this study does not take advantage of SSE/SSE2 SIMD extensions, reportedly because of the lack of a vector maximum operation we mentioned earlier. Wun et al. also provide a basic workload characterization of HMMer and a scalability analysis for both JackHMMer and a modified local memory version of JackHMMer. This local memory version works by copying models to very fast local ME memory, which bears a similarity to SPE local stores. The authors found that this version performs 3.5x times faster than the x86 baseline and scales much better to more than 15 microengines, but is limited to small profile models due to the size of the ME local memories.

Horn et al.[44] implemented ClawHMMER, a version of HMMer for clusters of Nvidia or ATI graphics processors(GPUs). Using their GPU-specific HMMER implementation written in the Brook programming language, the authors were able to obtain an almost 37x speedup over an unmodified HMMER running on a 2.8GHz Intel Pentium 4 CPU, and a 3.9x speedup over the optimized AltiVec version by Lindahl running on a 2.5GHz PowerPC G5. They also demonstrate linear speedup on a 16-node GPU rendering cluster. This work by Horn et al. also classify Cell BE as a streaming processor architecture not unlike the GPUs they used in their study, and suggest that a similar approach to theirs for parallelizing HMMER on GPUs would be suitable for a future Cell BE implementation. While our programming model differed somewhat from the one outlined in this paper, we found the discussion of their batching strategies useful for early guidance

of our efforts.

Cell-HMMER differs from both JackHMMER and ClawHMMER in a few critical aspects. ClawHMMER converts the Plan7 Viterbi algorithm into a streaming workload; while Cell-HMMER maintains the structure of the original Viterbi code in HMMER. We plan to investigate the performance of a streaming HMMER on the Cell BE as part of out future work. JackHMMER is essentially an implementation of *hmmpfam*, and utilizes a different axis of parallelism than the one used in our work. As mentioned earlier, the strategy we used to port HMMER to the Cell BE could not be used for *hmmpfam* due to the need to maintain the complete intermediate probability matrix for this application.

The Cell BE multiprocessor was first described in [79], and later in more detail by Kahle et al. [48]. Kahle et al. explain the motivation for many Cell BE design decisions and provide a comprehensive discussion of Cell BE programming models. Another reference on Cell BE programming models is [60]. A later paper by Gschwind et al.[42] describes the design philosophy behind the Cell BE microarchitecture and includes a detailed discussion of the ”synergistic processing” computational model of the Cell BE SPEs. Kistler et al. [50] describe the Cell BE on-chip communication networks, protocols and related performance issues.

A recent and very comprehensive work by Eichenberger et al.[32] presents a detailed discussion of important compilation issues for the Cell BE.(An earlier work describing the same compiler technology is [33].) The authors describe a modern optimizing compiler framework that can not only apply Cell BE-specific optimizations like automatic memory alignment and SIMDization of sequential code, but also compile and automatically partition unmodified OpenMP programs to the Cell BE PPE and SPEs. The

auto-SIMDization optimizations applied by this compiler results in an average of 9.9x speedup over sequential code, and task-level automatic partitioning and parallelization yields an average of 6.8x speedup for a variety of benchmarks. While the current version of their compiler can not apply all of these optimization techniques at the same time, a future integrated implementation of this research compiler could offer significant performance improvements since SIMDization and automatic partitioning are techniques that are fairly independent of each other.Considering the difficulty of manual partitioning to program the Cell BE, we anticipate such compilers to be very important for widespread adoption of the Cell BE architecture.

Williams et al.[98] evaluated the performance of Cell BE architecture running a variety of scientific computing applications. They present a detailed analysis of how the Cell BE performs for traditional HPC workloads dominated by floating-point operations. They compare the Cell BE results with results obtained on a Cray X1E vector machine, an AMD Opteron x86 server, and an Intel Itanium 2 VLIW system. Their findings illustrate the performance advantages of the Cell BE architecture: they report a speedup of 5.5x running 2D FFTs, and 37x running a stencil computation over the base and report speedups of up to 5.5x over an Opteron. In addition, Williams et al. compare the power efficiency of the Cell BE against the aforementioned systems, and propose an improved Cell BE processor with better double-precision floating point performance.

5.7 Conclusions

In this chapter, we presented Cell-HMMER, a high-performance implementation of HMMER search on the Cell Broadband Engine architecture. When executed using all 8 SPEs on the processor, Cell-HMMER outperforms current dual-core x86 processors by a comfortable margin in both single-processor (2 cores total) and 2-way SMP (4 cores total) configurations. We expect Cell-HMMER to be competitive against current and future quad-core x86 microprocessors as well.

The Cell BE architecture offers the potential of significant performance improvements for workloads that can utilize its features effectively. For many different types of workloads, this requires the presence of some data parallelism that can leverage the multiple SPEs. The fact that we were able to demonstrate speedup with a single SPE suggests that the combination of the low memory latency of the SPE local stores and the high memory bandwidth of the Cell BE architecture can be very effective in executing such workloads. Additional performance improvements are possible by exploiting finer-grained data parallelism through the use of SIMD instructions, as we have done for developing Cell-HMMER. Judging by the results of previous work that used SIMD instructions to accelerate HMMER ([44],[96]) and our tests, we believe a per-core speedup of 2x to 4x could be attributed to the use of SIMD instructions in Cell-HMMER (The previously mentioned approximate speedup figures for Lindahl's AltiVec SIMD HMMER implementation are higher because it can process 8 states at a time, compared to 4 in Cell-HMMER). The rest of the speedup reflects the benefits of moving the data closer to the SPEs and using the low-latency local stores.

Most of the effort in porting HMMER to the Cell BE architecture was spent in designing and implementing an efficient PPE-SPE partitioning for the program logic, designing efficient data structures for DMA transfers between the main memory and the SPE local stores, and fine-tuning the synchronization mechanism. While our results justified the effort and time spent in the development of Cell-HMMER, we believe there is a need for more advanced development tools and methodologies for Cell BE development. Development of tools that could automate the partitioning work and reduce the difficulty of Cell BE programming. Integrated with compiler technologies that can convert scalar code to SIMD, it might be possible to use such compilers ([33]) and tools to produce a quick Cell BE port of suitable applications that can be further refined and optimized.

Considering the time and effort currently required to port applications to the Cell BE, equally needed are methodologies to rapidly assess whether a workload will benefit from the Cell BE architecture or not. Our choice of HMMER was guided by detailed analysis of the source code of the function that dominated its execution time, and lessons learned from previous work that implemented this workload on other architectures. However, such information may not readily be available for many workloads; and assessing the suitability of an application to the Cell BE multiprocessor remains a length process which often involves some exploratory coding. In our experience, relying solely on characterization data obtained on different platforms turned out to be a suboptimal method for such assessment tasks: Some of our earlier work involved preliminary research on a possible port of BLAST to the Cell BE architecture. At first sight, BLAST looks like it can benefit from the Cell BE architecture: Based on our studies, BLAST had relatively few branches (Figure 3.1), highest degree of branch predictability among BioBench benchmarks (Fig-

ure 3.4), and much lower cache miss rates than those of HMMER (Figures 3.5 and 3.6). Furthermore, about 90 percent of the execution time is spent in two or three functions in both DNA and protein variants of the BLAST workload (Figure 3.7). However, exploratory coding and analysis of the code revealed that BLAST performance was heavily dependent on fast I/O, and a Cell BE implementation of BLAST was not likely to result in performance improvements of the magnitude we were able to obtain for HMMER. It should be noted that other bioinformatics applications, such as the rest of the applications in BioBench, could also benefit from being ported to the Cell BE architecture. To this end, we expect to see and take part in more exploratory porting studies to investigate the potential Cell BE performance of these applications in the future.

As mentioned earlier, we observed speedups with even a single SPE. We believe that a single SPE-like execution unit with sufficient local storage could be a worthwhile addition to many conventional microarchitectures, and intend to expand our research in this direction. The Viterbi algorithm is used in many other applications such as speech processing, machine learning and telecommunications. We believe it should be possible to adapt our Cell BE implementation of the Viterbi algorithm to accelerate such applications, and to use similar approaches on aforementioned SPE-like accelerators in future processors.

We employed numerous techniques to improve the performance of Cell-HMMER to its current level. Cumulatively, these techniques such as aggressive SIMD optimization, double buffering and doing tracebacks on the PPE allowed us to utilize the the high on-chip memory bandwidth and the multi-level software managed memory architecture of the Cell BE to good effect. We still believe there is room for improvement: as the Cell BE

architecture gains acceptance and Cell BE development tools become more powerful and widely available, we hope to be able to continue working on improving Cell-HMMER. Several objectives of our future work are already well-defined. The size of SPE local stores limits the size of input HMMs Cell-HMMER can currently handle, and we would like to revisit this problem in the future. Future Cell BE variants will probably provide larger SPE local store sizes and dampen the impact of this limitation. In addition, we want to improve the IPC figures we are getting on the SPEs, primarily by optimizing the SPE portion of Cell-HMMER to eliminate residual branch misses in the code.

The Cell BE architecture offers significant performance benefits at the cost of some degree of programming complexity. For many workloads such as HMMER, the additional time required to successfully develop and debug Cell BE applications is worth the effort.

[1] http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html.

[2] http://paup.csit.fsu.edu.

[3] http://perfsuite.ncsa.uiuc.edu.

[4] http://www.sgi.com/industries/sciences/chembio/pdf/bioperf01.pdf.

[5] Architecting the era of tera. ftp://download.intel.com/technology/comms/nextnet/download/Tera_Era.pdf.

[6] DeCypher HMM Solution. http://www.timelogic.com/decypher_hmm.html.

[7] Extending the world's most popular processor architecture. ftp://download.intel.com/technology/architecture/new-instructions-paper.pdf.

[8] Sun briefing in computational biology - a technical white paper.

[9] M. D. Adams, J. M. Kelley, J. D. Gocayne, M. Dubnick, M. H. Polymeropoulos, H. Xiao, C. R. Merril, A. Wu, B. Olde, and R. F. Moreno. Complementary DNA sequencing: expressed sequence tags and human genome project. *Science*, 252(5013):1651–1656, 1991.

[10] T. Agerwala and S. Chatterjee. Computer architecture: Challenges and opportunities for the next decade. *IEEE Micro*, pages 58–69, May/June 2005.

[11] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of International Symposium on Performance Analysis of Systems and Software(ISPASS)*, March 2005.

[12] S. F. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[13] D. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Proceedings of IEEE International Symposium on Workload Characterization(ISWC)*, October 2005.

[14] D. Bader, V. Sachdeva, V. Agarwal, G. Goel, and A. N. Singh. BioSPLASH: A sample workload for bioinformatics and computational biology for optimizing next-generation performance computer systems. Technical report, University of New Mexico, 2005.

[15] T. Barrett, T. O. Suzek, D. B. Troup, S. E. Wilhite, W.-C. Ngau, P. Ledoux, D. Rudnev, A. E. Lash, W. Fujibuchi, and R. Edgar. NCBI GEO: mining millions of expression profilesdatabase and tools. *Nucleic Acids Research*, 33(Database Issue):D562–D566, 2005.

[16] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, pages 3–14, June 1998.

[17] A. Bateman, L. Coln, R. Durbin, R. D. Finn, V. Hollich, S. Griffith-Jones, A. Khanna, M. Marshall, S. Moxon, E. L. L. Sonnhammer, D. J. Studholme, C. Yeats, and S. R. Eddy. The Pfam protein families database. *Nucleic Acids Research*, 32(Database issue):D138–D141, 2004.

[18] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. GenBank. *Nucleic Acids Research*, 34(Database Issue):D16–D20, 2006.

[19] H. M. Berman, J. Westbrook, Z. Feng, Gary Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

[20] E. Birney. Hidden Markov models in biological sequence analysis. *IBM Journal of Research and Development*, 45(3/4):449–454, May/July 2001.

[21] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A portable programming interface for performance evaluation on modern processors. Technical Report UT-CS-00-445, University of Tennessee, Dept. of Computer Science, July 2000.

[22] D. Caliga and D. R. Barker. Delivering acceleration: the potential for increased HPC application performance using reconfigurable logic. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, 2001.

[23] U. Catalyurek, E. Stahlberg, R. Ferreira, T. Kurc, and Joel Saltz. Improving performance of multiple sequence alignment analysis in multi-client environments. In *Online Proceedings of the First International Workshop on High Performance Computational Biology (HICOMB)*, 2002.

[24] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, pages 32–45, May/June 2006.

[25] G. Cochrane, P. Aldebert, N. Althorpe, M. Andersson, W. Baker, A. Baldwin, K. Bates, S. Bhattacharyya, P. Browne, A. van der Broek, M. Castro, K. Duggan, R. Eberhardt, N. Faruque, J. Gamble, C. Kanz, T. Kulikova, C. Lee, R. Leinonen, Q. Lin, V. Lombard, R. Lopez, M. McHale, H. McWilliam, G. Mukherjee, F. Nardone, M. P. Pastor, S. Sobhany, P. Stoehr, K. Tzouvara, R. Vaughan, R. Wu, W. Zhu, and R. Apweiler. EMBL nucleotide sequence database: developments in 2005. *Nucleic Acids Research*, 34(Database Issue):D10–D15, 2006.

[26] J. Cohen. Bioinformatics-an introduction for computer scientists. *ACM Computing Surveys*, 36(2):122–158, 2004.

[27] International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, 2004.

[28] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP throughput with mediocre cores. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques(PACT)*, pages 51–62, 2005.

[29] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[30] S. Eddy. *HMMER(Version 2.3.2) User's Guide*. HHMI/Washington University School of Medicine, 2003.

[31] S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[32] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[33] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for a CELL processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques(PACT)*, pages 161–172, 2005.

[34] P. L. Elkin. Primer on medical genomics part v: Bioinformatics. *Mayo Clinic Proceedings*, 78:57–64, January 2003.

[35] J. Felsenstein. PHYLIP-phylogeny inference package (version 3.2). *Cladistics*, 5:164–166, 1989.

[36] D.-F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–260, 1987.

[37] A. Finkelstein, J. Hetherington, L. Li, O. Margoninski, P. Saffrey, R. Seymour, and A. Warner. Computational challenges of systems biology. *IEEE Computer*, 37(5):26–33, 2004.

[38] M. J. Flynn and P. Hung. Microprocessor design issues: thoughts on the road ahead. *IEEE Micro*, pages 16–31, May/June 2005.

[39] P. L. Freddolino, A. S. Arkhipov, S. B. Larson, A. McPherson, and K. Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure*, 14:437–449, March 2006.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[41] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof:a call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982.

[42] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, pages 10–24, March/April 2006.

[43] D. G. Higgins and P. M. Sharp. CLUSTAL:a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73:237–244, 1988.

[44] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A streaming HMMer-Search implementation. In *Proceedings of The 2005 ACM/IEEE Conference on Supercomputing*, November 2005.

[45] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (llc) performance of data mining workloads on a CMP– a case study of parallel bioinformatics workloads. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006.

[46] D. T. Jones, R. T. Miller, and J. M. Thornton. Successful protein fold recognition by optimal sequence threading validated by rigorous blind testing. *Proteins*, 23:387–397, 1995.

[47] P. B. C. Jones. The commercialization of bioinformatics. In *Electronic Journal of Biotechnology*, volume 3, 2000.

[48] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[49] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, pages 66–76, March/April 2003.

[50] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, pages 10–23, May/June 2006.

[51] K. Koch and P. Henning. Beyond a single cell. In *Summit on Software and Algorithms for the Cell Processor*, 2006. `http://www.cs.utk.edu/~dongarra/cell2006/cell-slides/04-Ken-Koch.pdf`.

[52] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler. Hidden markov models in computational biology. *Journal of Molecular Biology*, 235:1501–1531, 1994.

[53] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, pages 64–75, June 2004.

[54] R. Kumar, D. M. Tullsen, N. P. Jouppi, and R. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, pages 32–38, November 2005.

[55] J. Landman, J. Ray, and J. P. Walters. Accelerating HMMER searches on Opteron processors with minimally invasive recoding. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA)*, pages 289–294, 2006.

[56] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture(MICRO)*, pages 330–335, 1997.

[57] Y. Li, T. Li, T. Kahveci, and J. A. B. Fortes. Workload characterization of bioinformatics applications. In *Proceedings of IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems(MASCOTS)*, October 2005.

[58] C.-K Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2005.

[59] N. M. Luscombe, D. Greenbaum, and M. Gerstein. What is bioinformatics? an introduction and overview. *IMIA Yearbook of Medical Informatics 2001*, pages 83–99, 2001.

[60] D. Mallinson and M. DeLoura. CELL:a new platform for digital entertainment. In *Online Proceedings of Game Developers Conference (GDC)*, 2005.

[61] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[62] J. M. May. MPX:software for multiplexing hardware performance counters in multithreaded programs. In *Proceedings of IEEE and ACM International Parallel and Distributed Processing Symposium(IPDPS)*, page 22, April 2001.

[63] X. Meng and V. Chaudhary. Bio-sequence analysis with Cradle's 3SoC scalable system on a chip. In *Proceedings of the ACM Symposium on Applied Computing*, pages 202–206, 2004.

[64] N. M. Morel, J. M. Holland, J. v.d. Greef, E. W. Marple, C. Clish, J. Loscalzo, and S. Naylor. Primer on medical genomics part xiv: Introduction to systems biology-a new approach to understanding disease and treatment. *Mayo Clinic Proceedings*, 79:651–658, May 2004.

[65] K. Muriki, K. D. Underwood, and R. Saas. RC-BLAST: Towards a portable, cost-effective open source hardware implementation. In *Online Proceedings of the Fourth International Workshop on High Performance Computational Biology (HICOMB)*, April 2005.

[66] O. Mutlu and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, 2003.

[67] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[68] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[69] C. Notredame, D. Higgins, and J. Heringa. T-Coffee: A Novel method for multiple sequence alignments. *Journal of Molecular Biology*, 302:205–217, 2000.

[70] M. Oka and M. Suzuoki. Designing and programming the emotion engine. *IEEE Computer*, 19(6):20–28, 1999.

[71] K. Okubo, H. Sugawara, T. Gojobori, and Y. Tateno. DDBJ in preparation for overview of research activities behind data submissions. *Nucleic Acids Research*, 34(Database Issue):D6–D9, 2006.

[72] T. Oliver, B. Schmidt, and D. Maskell. Hyper customized processors for bio-sequence database scanning on FPGAs. In *Proceedings of the ACM/SIGDA In-*

ternational Symposium on Field-Programmable Gate Arrays (FPGA), pages 229–237, 2005.

[73] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, 1996.

[74] J. L. Ortega-Arjona and G. Roberts. Architectural patterns for parallel programming. In *Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing*, 1998.

[75] R. D. M. Page and E. C. Holmes. *Molecular evolution: a phylogenetic approach*. Blackwell Science, 1998.

[76] M. J. Pallen. Microbial genomes. *Molecular Microbiology*, 32:907–912, 1999.

[77] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85(8):2444–2448, April 1988.

[78] J. L. Peterson, P. J. Bohrer, L. Chen, E. N. Elnozahy, A. Gheith, R. H. Jewell, M. D. Kistler, T. R. Maeurer, S. A. Malone, D. B. Murrell, N. Needel, K. Rajamani, M. A. Rinaldi, R. O. Simpson, K. Sudeep, and L. Zhang. Application of full-system simulation in exploratory system design and development. *IBM Journal of Research and Development*, 50(2/3):321–332, March/May 2006.

[79] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak,

M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, pages 184–185, February 2005.

[80] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26:1781–1802, 2005.

[81] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, February 1989.

[82] T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[83] B. Rost and S. O'Donoghue. Sisyphus and prediction of protein structure. *Computer Applications in the Biosciences*, 13:345–356, 1997.

[84] S. Sair and M. Charney. Memory behavior of the spec2000 benchmark suite. Technical Report RC-21852, IBM T.J. Watson Research Center, October 2000.

[85] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[86] S. Sohoni, Z. Xu, R. Min, and Y. Hu. A study of memory performance of multimedia applications. In *Proceedings of ACM International Conference on Mea-*

surement & Modeling of Computer Systems(SIGMETRICS), pages 206–215, June 2001.

[87] SPEC. *SPEC Benchmark Suite Release 1.0*. SPEC, 1989.

[88] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 248–252, February 2005.

[89] B. Sprunt. Managing the complexity of performance monitoring hardware: The brink and abyss approach.

[90] H. Steen and M. Mann. The ABC's (and XYZ's) of peptide sequencing. *Nature Reviews Molecular Cell Biology*, 5:699–711, 2004.

[91] G. G. Sutton, O. White, M.D. Adams, and A.R. Kerlavage. Tigr assembler : A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1(2):9–19, 1995.

[92] J. D. Thompson, D. G. Higgins, and T.J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple-sequence alignment through sequence weighting positions-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, June 1994.

[93] TPC. *TPC Benchmark A*. Itom International Co., 1989.

[94] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Pro-*

ceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA), pages 250–260, February 1997.

[95] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, IT-13(2):260–269, April 1967.

[96] J. P. Walters, B. Qudah, and V. Chaudhary. Accelerating the HMMER sequence analysis suite using conventional processors. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA)*, pages 628–636, 2006.

[97] Y. Wang, J. B. Anderson, J. Chen, L. Y. Geer, S. He, D. I. Hurwitz, C. A. Liebert, T. Madej, G. H. Marchler, A. Marchler-Bauer, A. R. Panchenko, B. A. Shoemaker, J. S. Song, P. A. Thiessen, R. A. Yamashita, and S. H. Bryant. MMDB: Entrezs 3D-structure database. *Nucleic Acids Research*, 30(1):249–252, 2002.

[98] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers (CF)*, pages 9–20, 2006.

[99] T. Wolf and M. Franklin. Commbench - a telecommunications benchmark for network processors. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*, pages 154–162, 2000.

[100] W. Wulf and S. McKee. Hitting the memory wall:implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, 1995.

[101] B. Wun, J. Buhler, and P. Crowley. Exploiting coarse-grained parallelism to accelerate protein motif finding with a network processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques(PACT)*, September 2005.

[102] T. K. Yap, O. Frieder, and R. L. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):283–294, 1998.